EFFICIENT ALGORITHMS AND DATA STRUCTURES FOR VLSI CAD

By

SEONGHUN CHO

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

1996

ACKNOWLEDGMENTS

My heartfelt appreciation goes to my advisor Professor Sartaj Sahni for giving me continued guidance in my thesis work. I thank him for the help, patience and support he provided throughout my stay in the University of Florida. Weekly meetings and discussions with him have spawned many ideas, for which I am thankful.

I would like to thank other members in my supervisory committee, Dr Li-Min Fu, Dr Theodore Johnson, Dr Sanguthevar Rajasekaran, and Dr Paul W. Chun for their interest and comments.

Thanks go to Venkat Thanvantri for his willingness to discuss the general subject of algorithms.

Thanks go to my wife Joungyim for her love, encouragement and patience. Finally, I would like to thank my parents for the love and support, without which I could not have pursued my doctoral studies. To them I dedicate this work.

TABLE OF CONTENTS

ACKNOWLEDGMENTS ii

LIST OF TABLES vii

LIST OF FIGURES x

ABSTRACT xi

CHAPTERS

1 INTRODUCTION 1

1.1 Background 1
1.2 Dissertation Outline 2

2 MINIMUM AREA JOINING OF COMPACTED CELLS 4

2.1 Introduction 4
2.2 l-layer River Routing 9
2.3 Constraint Graph Representation 16
2.4 Heuristics to Minimize Area 20
2.4.1 Heuristic 1 20
2.4.2 Heuristic 2 21
2.4.3 Heuristic 3 23
2.5 Experimental Results 25
2.6 Conclusion 30

3 A NEW WEIGHT BALANCED BINARY SEARCH TREE 32

3.1 Introduction 32
3.2 Balanced Trees and Rotations 34
3.3 β-BBSTs 39
3.4 Search, Insert, and Delete in a β-BBST 44
3.4.1 Search 44
3.4.2 Insertion 44
3.4.3 Deletion 57
3.4.4 Enhancements 63
3.4.5 Top Down Algorithms 64

3.5 Simple β-BBSTs 66
3.6 BBSTs without Deletion 70
3.7 Experimental Results 72
3.8 Conclusion 98

4 WEIGHT BIASED LEFTIST TREES AND MODIFIED SKIP LISTS 105

4.1 Introduction 105
4.2 Weight Biased Leftist Trees 106
4.3 Modified Skip Lists 109
4.4 MSLs As Priority Queues 118
4.5 Experimental Results For Priority Queues 122
4.6 Conclusion 129

5 CONCLUSIONS 135

A ABBREVIATIONS 136

REFERENCES 138

BIOGRAPHICAL SKETCH 141

LIST OF TABLES

2.1 Error rate (%) over optimal, $l = 1$ 27

2.2 Improvement (%) over Fang, $l = 1$ 27

2.3 Time taken, $l = 1$ 28

2.4 Error rate (%) over optimal, $l = 2$ 29

2.5 Improvement (%) over $l = 1$ cases 29

2.6 Improvement (%) over Fang, $l = 2$ 30

2.7 Time taken, $l = 2$ 31

3.1 The number of key comparisons on random inputs (version 1 code) . 76

3.2 The number of key comparisons on ordered inputs (version 1 code) . 77

3.3 Height of the trees on random inputs (version 1 code) 77

3.4 Height of the trees on ordered inputs (version 1 code) 77

3.5 The number of rotations on random inputs (version 1 code) 79

3.6 The number of rotations on ordered inputs (version 1 code) 80

3.7 Run time on random inputs using integer keys (version 1 code) 81

3.8 Run time on ordered inputs using integer keys (version 1 code) 82

3.9 Run time on random real inputs (version 1 code) 83

3.10 Run time on ordered real inputs (version 1 code) 84
3.11 The number of key comparisons on random inputs (version 1 code) . 87
3.12 The number of key comparisons on ordered inputs (version 1 code) . 88
3.13 Height of the trees on random inputs (version 1 code) 89
3.14 Height of the trees on ordered inputs (version 1 code) 89
3.15 The number of rotations on random inputs (version 1 code) 90
3.16 The number of rotations on ordered inputs (version 1 code) 91
3.17 Run time on random inputs using integer keys (version 1 code) 93
3.18 Run time on ordered inputs using integer keys (version 1 code) 94
3.19 Run time on random real inputs (version 1 code) 95
3.20 Run time on ordered real inputs (version 1 code) 96
3.21 The number of key comparisons on random inputs (version 2 code) . 99
3.22 The number of key comparisons on ordered inputs (version 2 code) . 100
3.23 Run time on random real inputs (version 2 code) 101
3.24 Run time on ordered real inputs (version 2 code) 103
4.1 The number of key comparisons 117
4.2 Number of levels 118
4.3 Run time 119
4.4 The number of key comparisons 124
4.5 Height/level of the structures 126
4.6 Run time using integer keys 127
4.7 The number of key comparisons 130

4.8 Height/level of the structures 131

4.9 Run time using integer keys 132

A.1 Abbreviations used in tables 137

LIST OF FIGURES

2.1 Cell joining 5

2.2 l-layer river routing 8

2.3 Round robin and greedy layer assignments 11

2.4 Minimizing the number of tracks or layers 14

2.5 Constraint graph representation 18

2.6 Merge in constraint graph 19

2.7 Heuristic 1 20

2.8 Heuristic 2 22

2.9 Heuristic 3 24

3.1 LL and RL rotations 35

3.2 A tree in WB(1/4) that is not $\frac{1}{3}$-balanced 42

3.3 $\frac{1}{2}$-balanced tree that is not a COST 43

3.4 LL rotation for insertion 45

3.5 Substep (i) of insertion LR rotation 46

3.6 Case LL for LR(ii) rotation 49

3.7 Case LR for LR(ii) rotation 51

3.8 LL rotation for deletion 58
3.9 LR rotation for deletion 60
3.10 Restructuring procedure 65
3.11 Simple restructuring procedure for insertion 68
3.12 Simple restructuring procedure for deletion 68
3.13 Simple restructuring procedure without a β value 70
3.14 Run time on real inputs (version 1 code) 85
3.15 Run time on random real inputs (version 1 code) 92
3.16 Run time on ordered real inputs (version 1 code) 97
3.17 Run time on random real inputs (version 2 code) 102
3.18 Run time on ordered real inputs (version 2 code) 102
4.1 Example min-WBLTs 108
4.2 min-WBLT Insert 109
4.3 min-WBLT Delete-min 110
4.4 Skip Lists 111
4.5 Modified Skip Lists 113
4.6 MSL Search 113
4.7 MSL Insert 114
4.8 MSL Delete 114
4.9 Run time 120
4.10 TMSL Insert 121
4.11 TMSL Delete-min 121

4.12 TMSL Delete-max 122

4.13 Run time on random1 128

4.14 Run time on random2 129

4.15 Run time on random1 131

4.16 Run time on random2 133

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Doctor of Philosophy

EFFICIENT ALGORITHMS AND DATA STRUCTURES FOR VLSI CAD

By

Seonghun Cho

May 1996

Chairman: Dr. Sartaj Sahni
Major Department: Computer and Information Science and Engineering

In this dissertation, we develop efficient algorithms and data structures for problems that arise in electronic computer aided design (ECAD).

We consider the problem of joining a row of compacted cells so as to minimize the area occupied by the cells and the interconnects. The cell joining process includes cell stretching and river routing. We propose several heuristics to join a row of cells in such a way that area is minimized. The proposed heuristics are compared experimentally with the previously proposed heuristic.

We develop a new class of weight balanced binary search trees called β-balanced binary search trees (β-BBSTs). β-BBSTs are designed to have reduced internal path

length. As a result, they are expected to exhibit good search time characteristics. Individual search, insert, and delete operations in an n node β-BBST take $O(\log n)$ time for $0 < \beta \leq \sqrt{2} - 1$. Experimental results comparing the performance of β-BBSTs, WB(α) trees, AVL-trees, red/black trees, treaps, deterministic skip lists and skip lists are presented. Two simplified versions of β-BBSTs are also developed.

We propose the weight biased leftist tree as an alternative to traditional leftist trees for the representation of mergeable priority queues. A modified version of skip lists that uses fixed size nodes is also proposed. Experimental results show our modified skip list structure is faster than the original skip list structure for the representation of dictionaries. Experimental results comparing weight biased leftist trees and competing priority queue structures as well as experimental results for double ended priority queues are presented.

CHAPTER 1
INTRODUCTION

1.1 Background

In VLSI layout, we are concerned with transforming a circuit from its logical design to a physical implementation. The layout problem for VLSI circuits is generally decomposed into smaller problems such as partitioning, floorplanning, placement, routing and compaction.

The partitioning process decomposes a large circuit/module into a collection of smaller sub-circuits/modules. In floorplanning, logical components of a circuit are assigned relative positions on a chip. The physical realization of each component (i.e., its area and aspect ratio) is also selected. The objectives of floorplanning include overall area minimization, minimization of power consumption, etc. The precise locations for the components of a design are then determined during the placement process to optimize the area and the timing. After the components are placed, the pins are connected during the routing process. During the process of compaction, the components and interconnections are moved so as to further optimize the layout in terms of area and delay.

The routing process is usually divided into three smaller sub-problems of global routing, detailed routing, and specialized routing. Global routing decomposes the

complex routing problem into small and manageable sub-problems. It assigns each net to a set of routing regions such as channel, switchbox, and planar routing to minimize a combination of criteria such as area, circuit delay, etc. Steiner trees and spanning trees are the commonly used approaches for net connection in global routing. Specialized routing is used to connect power-ground nets or clock nets. Detailed routing has two types of routing, general and restricted and there are three types of detailed restricted routing, channel, switchbox, and planar.

In channel routing, all terminals of nets are located in two parallel rows across a routing region called channel. In switchbox routing, terminals of nets are located on the four sides of the routing region. Planar routing is a problem in which the interconnection topology of the nets is planar. That is, all connections can be realized on a single layer. Vias allow wires to change layers but the presence of vias reduces reliability and performance of a circuit. Single layer routing is not always possible.

River routing is a special case of planar routing in which all nets have exactly two terminals, one on each side of the channel, and the net sequence on each side of the channel is the same. River routing is used in PCB routing, particularly in dataflow architectures with multi-bit buses connecting a series of logic blocks, and symbolic IC design systems.

1.2 Dissertation Outline

This dissertation is divided into four chapters. In Chapter 2, we consider the problem of joining a row of compacted cells so as to minimize the area occupied by the cells and the interconnects. The cell joining process includes cell stretching and

river routing. We propose several heuristics to join a row of cells in such a way that area is minimized. The proposed heuristics are compared experimentally with the previously proposed one.

VLSI Physical Design Automation is essentially the study of algorithms and data structures related to the physical design process. Specific data structures can be used to improve the performance of algorithms. For example, maze routing algorithms and line-probe algorithms in global routing [30] use search structures, retiming algorithms [6, 29] use priority queue structures, and layout compaction algorithms [13] use radix priority search trees.

Chapters 3 and 4 are about search and priority queue structures. As specific data structures could be used to produce better performance of algorithms in VLSI design automation, new search and priority queue structures are proposed and thoroughly compared with other data structures.

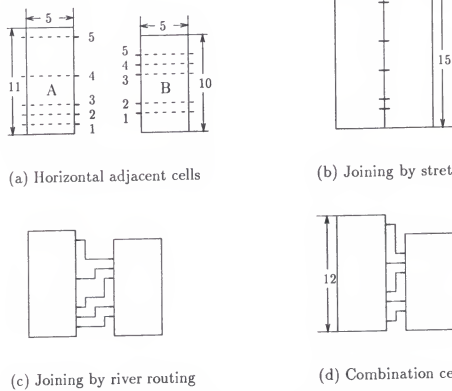
Finally, in the last chapter, we present conclusions of this work.

CHAPTER 2
MINIMUM AREA JOINING OF COMPACTED CELLS

2.1 Introduction

When designing circuits with compacted symbolic sticks basic cells, the circuit is realized by a collection of compacted cells that tile a two-dimensional area. The intercell interconnects are such that each interconnect connects two terminals that are on adjacent boundaries of neighboring cells. So, for example, if cells A and B (Figure 2.1(a)) are neighboring cells of the circuit, then the right boundary of A is adjacent to the left boundary of B. The number of terminals on each of these boundaries will be the same and the i'th terminal (from the bottom) on the right boundary of A is to be connected to the i'th terminal (from the bottom) on the left boundary of B.

Since the cells are available in compacted form, it is not possible to reduce the distance between any pair of terminals on any side of a cell. However, this distance can be increased by stretching the cell. In the example of Figure 2.1(a), we can stretch either cell vertically by defining a horizontal cut line at any position and pulling the two cell pieces apart by any desired amount (the cell can also be stretched horizontally by using a vertical cut line).



(a) Horizontal adjacent cells

(b) Joining by stretching

(c) Joining by river routing

(d) Combination cell joining

Figure 2.1. Cell joining

The required interconnects between cells A and B of Figure 2.1(a) can be accomplished by stretching cells A and B so that the terminals of A and B line up as in Figure 2.1(b). The broken lines in Figure 2.1(a) indicate the cut lines used for stretching. The stretching enables us to join cells A and B using no routing tracks (by "join" we mean make the interconnects between cells A and B). This method of joining cells is also called pitch matching.

Another way to join cells A and B is to river route the interconnects as in Figure 2.1(c). This uses routing tracks in a channel between cells A and B but does not increase cell height. The pitch matching and river routing approaches to cell joining have been studied in Boyer [5] and Weste [33]. Algorithms for single-layer

river routing can be found in several works [15, 19, 23, 24] and those for multilayer river routing can be found in Baratz [3]. Single-layer gridless river routing is studied in Tompa [32]. Two applications of river routing are hybrid circuit design and structured design (DSP).

Cell stretching (or pitch matching) increases the height of the layout while river routing increases its width. Both affect the layout area. The layout of Figure 2.1(b) has area 150. To compute the area of the layout of Figure 2.1(c), we assume tracks have unit separation. So, the layout width is 14 and height is 11. The layout has area 154. Cheng and Despain [8] have proposed using a combination of cell stretching and river routing so as to obtain layouts with smaller area than possible when only one of these joining methods is used. Figure 2.1(d) shows the result of joining cells A and B using both stretching and river routing. The area of this layout is 144. This is minimum for the instance of Figure 2.1(a).

Cheng and Despain [8] have proposed a heuristic for single layer joining of compacted cells. At each step of their heuristic either a row or column of compacted cells is joined. Following this, the row or column of joined cells is replaced by a composite cell that represents the result of joining. Notice that when a row (column) of cells is joined, cells may be stretched vertically (horizontally) and river routing is done in a vertical (horizontal) channel. To join a row of cells, Cheng and Despain [8] bound the maximum height to which a cell may be stretched. This bound is

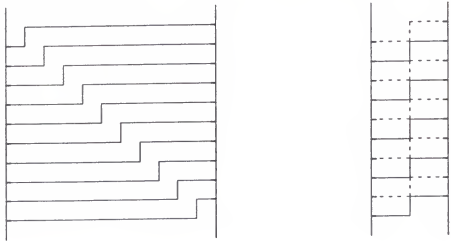
$$h_{max} + h_{avg}^2/(4 * h_{max})$$

where h_{max} is the height of the tallest compacted cell being joined and h_{avg} is the average height of the cells being joined.

Using this bound, cells are joined one-at-a-time using a penalty/reward scheme to determine if a pair of terminals is to be joined by stretching or by river routing.

Lim, Cheng, and Sahni [17] have considered the case when only two cells are to be joined. They develop fast polynomial time algorithms to obtain the minimum area join of two cells. In addition, they are able to obtain, in low order polynomial time, minimum area joins that minimize the length of the longest wire or the total wire length. Lim [16] has proposed an $O(n(n/c)^{c-1})$ algorithm to find the minimum area join of c cells having a total of n terminals. This algorithm does an exhaustive search over all possible numbers of tracks in the $c-1$ routing channels between adjacent cells. A constraint graph is used to determine the minimum height layout for each assignment of number of tracks to routing channels. The time required per track assignment is $O(n)$ and the worst case number of track assignments is $O((n/c)^{c-1})$. The algorithm of Lim [16] is flawed as it handles channels with zero routing tracks by joining the adjacent cells using minimum height cell stretching and then considers the joined cells as one. This problem is easily fixed, however, by combining, in the constraint graph, pairs of vertices that represent corresponding terminals of the two cells (i.e., i'th terminals of each cell) with zero routing tracks in between.

In this chapter, we consider the case when $l \geq 1$ routing layers are available to river route the inter cell connections. Note that while multiple layers do not affect layout area when cell stretching alone is used, a reduction in area is possible when



(a) 1-layer (b) 2-layer

Figure 2.2. l-layer river routing

cell stretching is combined with river routing or when river routing alone is used. We assume that in each layer of each routing channel, the interconnects are to be accomplished using river routing. An alternative is to use HV routing when $l = 2$, HVH or VHV routing when $l = 3$, and extensions of HVH and VHV routing for $l > 3$. However, for river routing instances, using routing layers in this way has no advantage over river routing in each layer (see Theorem 5, Section 2). When the number of layers available for river routing is increased, one may see a dramatic reduction in the number of routing tracks needed per layer. Figure 2.2 shows an instance that needs n tracks when routed in one layer but only one track/layer when routed in two layers.

We begin, in section 2, by stating the necessary and sufficient conditions for a river routing instance to be routable in l layers using at most t tracks per layer and stating how to perform l-layer river routing when such a routing is possible. In

this section, we also show that HV style routing has no advantage over river routing in each layer. In Section 2.3, we describe the constraint graph used to determine minimum height stretching of c cells. Heuristics for the minimum area joining of c cells are proposed in Section 2.4 and the results of experiments with these are provided in Section 2.5. Our conclusions appear in Section 2.6.

2.2 l-layer River Routing

Let (A_i, B_i), $1 \leq i \leq m$, be a set of terminal pairs such that the A_i's are on one side (say left or top) of a routing channel and the B_i's are on the other (right or bottom) side. Terminal A_i is to be connected to terminal B_i, $1 \leq i \leq m$. For this channel routing instance to be an instance of river routing, it must be the case that $a_1 < a_2 < \ldots < a_m$ and $b_1 < b_2 < \ldots < b_m$ where a_i and b_i, respectively, give the positions of terminals A_i and B_i, $1 \leq i \leq m$. We may assume an underlying grid with each terminal being at a grid position. In the case of a horizontal (vertical) channel the a_is and b_is are grid column (row) numbers. Leiserson and Pinter [15] have obtained the following necessary and sufficient condition for a river routing instance to be routable in a single layer using at most $t \geq 0$ tracks.

Theorem 1 [15] *The river routing instance defined above is routable in a single layer using at most $t \geq 0$ tracks if and only if*

(a) $a_{i+t} - b_i \geq t$

(b) $b_{i+t} - a_i \geq t$

for every $i \leq m - t$.

For the general case of $l \geq 1$ layers, we obtain the necessary and sufficient condition of Theorem 2.

Theorem 2 *The river routing instance defined above is routable in $l \geq 1$ layers (each layer routing whole nets) using at most $t \geq 0$ tracks per layer if and only if*

(a) $a_{i+lt} - b_i \geq t$

(b) $b_{i+lt} - a_i \geq t$

for every $i \leq m - lt$.

Proof: First, we establish that (a) and (b) are necessary for l layer routing. Since the proofs for (a) and (b) are similar, we provide that for (a) only. Suppose that $a_{i+lt} - b_i < t$ for some i. Consider the $lt+1$ terminal pairs (A_j, B_j), $i \leq j \leq i + lt$. When routing these on l layers, at least one layer has to be assigned $\geq t+1$ terminal pairs. So, suppose that terminal pairs (A'_1, B'_1), (A'_2, B'_2), ..., (A'_{t+1}, B'_{t+1}), ... are assigned to the same layer for river routin g. We may assume that $a'_1 < a'_2 < \ldots < a'_{t+1}$ and $b'_1 < b'_2 < \ldots < b'_{t+1}$. Since $a'_{t+1} \leq a_{i+lt}$ and $b'_1 \geq b_i$, $a'_{t+1} - b'_1 \leq a_{i+lt} - b_i < t$. From Theorem 1, it follows that the terminal pairs (A'_j, B'_j), $1 \leq j \leq t+1$, cannot be river routed on a single layer. Hence, (A_j, B_j), $i \leq j \leq i + lt$, cannot be river routed on l layers. So, (A_j, B_j), $1 \leq j \leq m$, cannot be river routed on l layers. As a result, (a) is a necessary condition.

To show that (a) and (b) are sufficient conditions for routability, we present two algorithms (RoundRobin and Greedy) that assign the nets to layers in such a way that each layer is river routable when both (a) and (b) are satisfied. The correctness

```
procedure RoundRobin ;
{ Assign the m nets to l layers. }
begin
    for i := 1 to m do
        assign net (A_i, B_i) to layer (i mod l) + 1 ;
end ;

procedure Greedy ;
{ Assign the m nets to l layers. }
begin
    for i := 1 to m do
        assign net (A_i, B_i) to layer q such that q is the smallest integer ≤ l
        for which the conditions of Theorem 1 are not violated on layer q
        (if there is no such q, then fail) ;
end ;
```

Figure 2.3. Round robin and greedy layer assignments

of these algorithms is established in Theorems 3 and 4, respectively. □

We later discovered that Baratz [3] has not only obtained the same condition but also proposed the same two algorithms for l-layer river routing. One assigns nets to layers in a round robin fashion and the other uses a greedy strategy. The corresponding procedures are given in Figure 2.3.

Theorem 3 *The layer assignment produced by the RoundRobin procedure is river routable if*

(a) $a_{i+lt} - b_i \geq t$

(b) $b_{i+lt} - a_i \geq t$

for all $i \leq m - lt$.

Proof: Let $(A'_1, B'_1), (A'_2, B'_2), \ldots = (A_j, B_j), (A_{j+l}, B_{j+l}), (A_{j+2l}, B_{j+2l}), \ldots$ be the nets assigned to layer $(j \bmod l) + 1$, $j \leq l$. So, $a'_i = a_{j+(i-1)l}$ and $b'_i = b_{j+(i-1)l}$. Hence,

$$\begin{aligned} a'_{i+t} - b'_i &= a_{j+(i+t-1)l} - b_{j+(i-1)l} \\ &= a_{j+(i-1)l+tl} - b_{j+(i-1)l} \\ &\geq t \ \text{(from (a))} \end{aligned}$$

Similarly, $b'_{i+t} - a'_i \geq t$. So, the layer assignment satisfies the conditions of Theorem 1 and is river routable using t tracks. □

Theorem 4 *If (a) $a_{i+lt} - b_i \geq t$ and (b) $b_{i+lt} - a_i \geq t$ for all $i \leq m - lt$, then procedure Greedy assigns nets to layers such that the assignment to each layer is routable using t tracks.*

Proof: If procedure Greedy is able to assign each of the m nets to a layer, then the layer assignments satisfy the conditions of Theorem 1 and so are routable using t tracks. Suppose the algorithm fails while trying to assign net (A_r, B_r) to a layer. At this time nets (A_i, B_i), $1 \leq i < r$, have been assigned to layers so as to satisfy the conditions of Theorem 1 and the assignment of net (A_r, B_r) to each of these layers violates these conditions. Consider first those layers, L_a, on which condition (a) is violated. For a layer $s \in L_a$, suppose that the assigned nets are $\ldots, (A'_{j-t}, B'_{j-t}), \ldots, (A'_{j-1}, B'_{j-1})$. Let $(A'_j, B'_j) = (A_r, B_r)$. Since $s \in L_a$, we have $a'_j - b'_{j-t} < t$. Now, if $b_{r-lt} \geq b'_{j-t}$, then $a_r - b_{r-lt} < t$ which violates condition (a) of this theorem. So, $b_{r-lt} < b'_{j-t}$. Since, $b_{r-lt} < b'_{j-t} < \ldots < b'_{j-2} < b'_{j-1}$, t of the $lt - 1$ nets

$(A_{r-lt+1}, B_{r-lt+1}), \ldots, (A_{r-1}, B_{r-1})$ have been assigned to layer $s \in L_a$. Consequently, the layers in L_a account for $t|L_a|$ of these $lt - 1$ nets.

In a similar way, we can show that the remaining $l - |L_a|$ layers account for another $t(l - |L_a|)$ of these nets. This gives us a total of tl nets, whereas we had only $tl - 1$. This contradiction implies that procedure Greedy cannot fail unless conditions (a) and (b) are not satisfied. □

Procedure RoundRobin is easily seen to have complexity of $O(m)$. A straightforward implementation of procedure Greedy will have complexity of $O(ml)$. However, by using priority search trees [18] the complexity can be reduced to $O(m \log l)$. In practice, since l is quite small, it is unlikely that the priority search tree implementation will run faster than the straightforward implementation in which the l layers are checked in sequence. The actual routing for all l layers can be done in $O(mt)$ time using the computed layer assignment and the single layer routing algorithm of Leiserson and Pinter [15].

Using Theorem 2, we can develop a linear time algorithm to determine the minimum number of tracks needed to route an instance in l layers as well as to find the minimum number of layers needed for a t track routing. The algorithm for the former is given in Figure 2.4. This figure also shows the changes needed in case t is given and we wish to determine the minimum number of layers. The correctness of the algorithm follows from that of Theorem 2 and the fact that t (or l) is increased only if the current t (l) is found to be infeasible. The complexity is $O(m)$ as neither

```
procedure MinimizeTracks ;  {or MinimizeLayers}
{ Determine the minimum number of tracks per layer (or minimum number
  of layers) needed for multilayer river routing }
begin
    t := 0 ;  {or l := 1}
    i := 1 ;
    while (i ≤ m − lt) do
        if (a_{i+lt} − b_i < t) or (b_{i+lt} − a_i < t)
        then t := t + 1  {or l := l + 1}
        else i := i + 1 ;
end ;
```

Figure 2.4. Minimizing the number of tracks or layers

i nor t (l) can exceed m. So, neither clause of the **if** statement can be executed more than $m-1$ times.

Using the multilayer river routing results of Baratz [3], one can trivially extend all the results of Lim, Cheng and Sahni [17] to the case of multilayer joining of compacted cells. So, the multilayer minimum area join of two compacted cells with m nets can be obtained in $O(m^2)$ time. If we wish to minimize the maximum wire length while keeping area minimum, the asymptotic time complexity is still $O(m^2)$. The total wire length can be minimized while keeping area minimum in $O(m^2 \log m)$ time.

In HV style routing, each routing layer is assigned a routing direction (either H or V). In an H (V) layer only horizontal (vertical) wire segments can be laid out. Horizontal segments on one layer connect to vertical segments, of the same net, on another layer by means of vias. In the case of river routing instances, one can see that there is no advantage to having more than two V-layers (i.e., two V-layers are sufficient to route all river routing instances).

Let $\mathrm{RR}(l,t)$ be the set of all river routing instances that can be routed in l layers, using t tracks per layer and using river routing in each layer. Let $\mathrm{HV}(l,t)$ be all river routing instances that can be routed using HV style routing, l layers, and t tracks per layer. Note that $\mathrm{HV}(l,t)$ includes instances routable with 0, 1, and 2 V-layers. Let $\mathrm{HVV}(l,t)$ be all river routing instances using HV style routing, $l-2$ H-layers, and 2 V-layers. Theorems 5 and 6 below hold for both the knock-knee [25] and directional HV models. Theorem 7 holds only for the directional model.

Theorem 5 $\mathrm{HV}(l,t) \subset \mathrm{RR}(l,t)$ *for every* $l > 1$ *and every* $t \geq 1$.

Proof: $\mathrm{HV}(l,t) \subseteq \mathrm{RR}(l,t)$ follows from a more general result obtained by Baratz [3]. Baratz [3] has shown that, for river routing instances, there is no advantage to using any routing scheme that wires a net on more than one layer. Since it is easy to construct river routing instances X such that $X \in \mathrm{RR}(l,t)$ and $X \notin \mathrm{HV}(l,t)$, it follows that $\mathrm{HV}(l,t) \subset \mathrm{RR}(l,t)$.

We provide a simpler proof of $\mathrm{HV}(l,t) \subseteq \mathrm{RR}(l,t)$. This proof will also establish our next result. We shall show that if X is a river routing instance such that $X \notin \mathrm{RR}(l,t)$, then $X \notin \mathrm{HV}(l,t)$. Hence, $\mathrm{HV}(l,t) \subseteq \mathrm{RR}(l,t)$.

Suppose that $X \notin \mathrm{RR}(l,t)$. From Theorem 2, it follows that $a_{i+lt} - b_i < t$ or $b_{i+lt} - a_i < t$ for some i. Suppose that $a_{i+lt} - b_i < t$ (the proof is similar when $b_{i+lt} - a_i < t$). So, $a_{i+lt} < b_i + t$. Since X is a river routing instance, at least nets $i+t, \ldots, i+lt$ intersect a vertical cut line drawn at a_{i+lt}. Hence, the density of X at a_{i+lt} is $\geq i + lt - (i+t) + 1 = (l-1)t + 1$. When HV style routing is used with

l, $l > 1$, layers, at most $l-1$ layers are available for horizontal routes. With t tracks per layer, densities of at most $(l-1)t$ can be accommodated. So, $X \notin \mathrm{HV}(l,t)$. □

Theorem 6 $\mathrm{HVV}(l,t) \subset \mathrm{RR}(l-1,t)$ *for every* $l > 2$ *and every* $t \geq 1$.

Proof: As in Theorem 5, suppose that $X \notin \mathrm{RR}(l-1,t)$. Let i be such that $a_{i+(l-1)t} - b_i < t$. The net density at $a_{i+(l-1)t}$ is $\geq (l-2)t + 1$. In HVV routing, two layers are V-layers. So only $l-2$ layers are available for horizontal segments. This is not enough as the horizontal segment density is $\geq (l-2)t + 1$ at $a_{i+(l-1)t}$. Hence, $X \notin \mathrm{HVV}(l,t)$.

One may easily construct river routing instances that are in $\mathrm{RR}(l-1,t)$ but not in $\mathrm{HVV}(l,t)$. □

Theorem 7 $\mathrm{RR}(2,t) \not\subset \mathrm{HV}(l,t) - \mathrm{HVV}(l,t)$ *for every* $l > 1$ *and every* $t \geq 1$.

Proof: Consider the RR instance $(a_1, b_1) = (1,2)$ and $(a_2, b_2) = (2,3)$. This is in RR(2,t) for every $t \geq 1$ but is not in $\mathrm{HV}(l,t) - \mathrm{HVV}(l,t)$ for any l. □

As remarked earlier, Theorem 7 holds only for the directional model. For the knock-knee model, one can show that $\mathrm{RR}(l,t) \subset \mathrm{HV}(l+1,t)$ for every $l \geq 1$ and every $t \geq 1$.

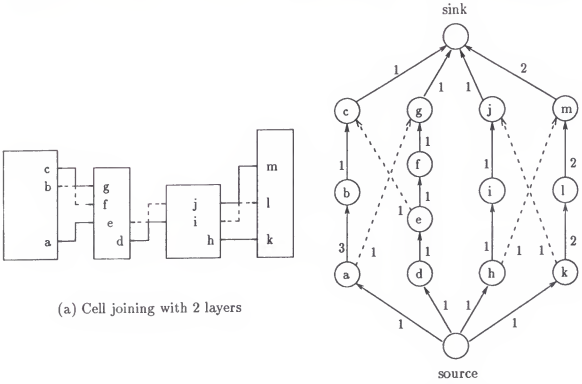
2.3 Constraint Graph Representation

Lim [16] has proposed the use of a constraint graph to determine the terminal positions in a row of compacted cells. This is for the case when the number of tracks

in each routing channel is given and we wish to minimize the layout height. In the constraint graph, each cell is represented by a directed chain of vertices. Each cell terminal is represented by a vertex. The exception is when a compacted cell has terminals at the same y-position on both sides of the cell. In this case, the two terminals at the same y-position are represented by a single vertex. The vertex chain is linked in the direction of increasing y-position. The chain edges are labeled by the minimum allowable terminal separation. In addition, the constraint graph contains a source vertex that represents the bottom of the layout and a sink vertex that represents the layout top. The source vertex connects to the bottom of each chain and the top of each chain is connected to the sink vertex.

Figure 2.5(b) shows the chains (solid edges) for the four cell row of Figure 2.5(a). To complete the constraint graph, directed edges are added to introduce the channel routing constraints of Theorem 2. These are represented by the broken edges of Figure 2.5(b). Figure 2.5(b) is for the two layer case.

Lim [16] has shown that the constraint graph is acyclic provided the number of tracks in each routing channel is > 0. He has proposed handling channels with zero tracks by finding first the minimum area joining of the adjacent cells (only cell stretching is permitted now) and then combining these two cells into one. I.e., the two cells are replaced by their minimum area join. This strategy can be shown to result in non-optimality of the algorithm proposed in Lim [16]. To preserve optimality, it is necessary to merge the vertices that represent terminals that are the endpoints of



(a) Cell joining with 2 layers

(b) Constraint graph representation

Figure 2.5. Constraint graph representation

nets that are to be routed using no tracks as in Figure 2.6. The resultant constraint graph is also acyclic.

It is easy to see that the number of vertices and edges in the constraint graph is $O(n)$ where n is the total number of terminals. Furthermore, the graph can be constructed in $O(n)$ time given the number of routing layers and the number of tracks in each channel. The constraint graph described by us is identical to that of Lim [16] except in the way channels with zero tracks are handled and in that our graph is defined for $l \geq 1$ routing layers while that of Lim [16] is only for $l = 1$.

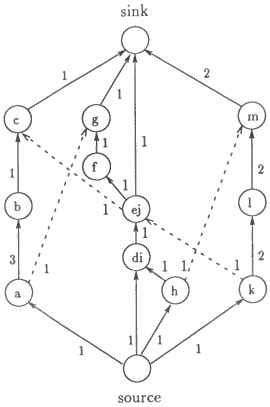


Figure 2.6. Merge in constraint graph

The length of the longest path from the source vertex of the constraint graph to each of the remaining vertices can be computed in $O(n)$ time by doing this in topological order [14, Section 6.5]. It is easy to see that if each terminal is placed at a vertical position given by the longest path length from the source, then all nets can be routed in the given number of tracks (as the conditions of Theorem 2 are satisfied in each routing channel). Furthermore, Lim [16] has shown that such a positioning of terminals results in a stretched layout of minimum height for the given channel widths. As a result, when channel widths are known, cells can be stretched to minimize area in $O(n)$ time. The channel widths that result in minimum area can be determined in $O(n(n/c)^{c-1})$ time where c is the number of cells by trying out all

```
procedure Heuristic1 ;
begin
    for i := 1 to c − 1 do
    begin
        determine the minimum area join of each pair of adjacent cells ;
        select the pair that has minimum area and replace it with its
        minimum area join ;
    end ;
end ;
```

Figure 2.7. Heuristic 1

possible channel widths [16]. Since this is feasible only for small c, we propose several heuristics in the next section.

2.4 Heuristics to Minimize Area

We formulate three greedy heuristics to obtain the minimum area join of a row of c compacted cells that have a total of n terminals.

2.4.1 Heuristic 1

The heuristic is described in Figure 2.7.

In each iteration of the **for** loop we examine every pair of adjacent cells. For each pair, the minimum area join is found using the algorithm of Lim, Cheng and Sahni [17] extended to the multilayer case as discussed in Section 2.2. The pair which has the minimum area join is replaced by a single cell that represents this join. So following each iteration of the **for** loop the number of cells decreases by one. When the **for** loop terminates, we are left with a single cell that represents the join of all c cells. The time needed to determine the minimum area join of a pair of cells with n_i nets between them is $O(n_i^2)$. The time to do this for all pairs of adjacent

cells is $O(\sum_i n_i^2) = O(n^2)$. So, the **for** loop iteration with $i = 1$ takes $O(n^2)$ time. On subsequent iterations, only the two pairs that include the cell introduced in the previous iteration need to have their minimum area join computed. Since each cell pair being considered includes at least one composite cell, the minimum area join is computed by considering the portion of the constraint graph that represents all the basic cells in the cell pair. Channel widths for channels within a composite cell are not changed while obtaining the minimum area join of the cell pair. However, as different channel widths for the channel between the two (composite) cells being joined are tried, the constraint graph is used to determine the minimum height of the combined cell. So, the time to combine two (composite) cells with n_i terminals in the channel between them is $O(nn_i)$. Hence the time for the remaining $c - 2$ iterations is $O(n \sum_i n_i) = O(n^2)$. The overall complexity of Heuristic 1 is therefore $O(n^2)$. In case the terminals are uniformly distributed over the cells, $n_i = O(n/c)$ for all i. The time for the first iteration of the **for** loop is now $O(n^2/c)$ and that for each of the remaining iterations is $O(n^2/c)$. The overall time is $O(n^2)$.

2.4.2 Heuristic 2

In this heuristic, we begin by assigning each channel the number of tracks needed to route the channel with no cell stretching. This number can be determined in $O(n_i)$ time for a channel with n_i nets as described in Section 2.2. The time taken to do this for all $c - 1$ channels is $O(n)$. The configuration obtained in this way is the maximum width layout. Starting from this configuration, we reduce the total number of tracks available across all $c - 1$ channels by one on each iteration. For this, the

```
procedure Heuristic2 ;
begin
    for each channel determine the number of tracks, t_i,
    needed to route with no stretching, 1 ≤ i < c ;
    t := Σ_{i=1}^{c-1} t_i ;
    set up the constraint graph using t_i tracks in channel i, 1 ≤ i < c ;
    compute layout area, A ;
    for tracks := t downto 1 do  {reduce by 1}
    begin
        for i := 1 to c − 1 do
        begin
            reduce the number of tracks in channel i by 1 ;
            modify the constraint graph to reflect this ;
            determine the length of the longest path in the graph
            and from this the layout area, a_i ;
        end ;
        select j such that a_j = min{ a_i } ;
        reduce the number of tracks in channel j by 1 ;
        A = min{ A, a_j } ;
    end ;
end ;
```

Figure 2.8. Heuristic 2

effect of a one track reduction is computed for each channel. The minimum layout height is determined by computing the length of the longest path in the constraint graph of Section 2.3. The track reduction is done in the channel that results in the smallest layout height (hence the minimum area for the given number of tracks). The algorithm is stated more formally in Figure 2.8.

When the algorithm terminates, A is the area of the minimum area join found by the heuristic. To reconstruct the layout, it is necessary to store the tracks per channel each time A is updated in the statement $A = min\{ A, a_j \}$.

For the time complexity, we see that the steps that precede the outer **for** loop take $O(n)$ time. Each iteration of the outer loop takes $O(nc)$ time. Hence this loop

contributes a total of $O(nct)$ to the time. Since $t = O(n)$, the overall time complexity of Heuristic 2 is $O(n^2c)$.

2.4.3 Heuristic 3

Unlike Heuristic 2 which attempts to minimize the layout height for each value of t, the total number of tracks, Heuristic 3 attempts to minimize the width (i.e., total number of tracks) for each choice of layout height. The heuristic begins with a layout height, ht, equal to the height of the tallest compacted cell. At each iteration, the next layout height to use is computed as described later. During each iteration, cells are combined in groups of at most k ($k > 1$ is a parameter to the heuristic). Each group of combined cells is replaced by its minimum area join subject to the constraint that the height of the join does not exceed ht. This joining of $\leq k$ cells at a time continues until only one cell remains. Its area is computed and recorded. The minimum area obtained over all heights tried is then reported as the best. Heuristic 3 is given in Figure 2.9.

In our implementation of Heuristic 3, the minimum area join of k cells is found by considering the portion of the constraint graph for all the basic cells included in these k cells. So, for this purpose composite cells are not handled as single cells. Rather, as in Heuristic 1, the basic cells they are composed of are considered and channel widths previously assigned to the associated channels are not changed. Track assignment is done only for the $k - 1$ channels between the k composite cells. We found this to give better results than when composite cells were regarded as atomic. For the case $k = 2$, the minimum area is determined by a binary search over the

```
procedure Heuristic3 ;
begin
    ht := height of the tallest cell ;
    repeat { minimize width subject to height ≤ ht }
        repeat { do this by combining k cells at a time }
            select k adjacent cells such that the minimum height cell is selected
            and the height of the tallest selected cell is minimum
            (if there are fewer than k cells, then select all of them) ;
            obtain the minimum area layout for the selected cells under
            the constraint that the layout height does not exceed ht ;
            during the preceding step record the next value of ht
            that is possible for a layout ;
        until one cell remains ;
        compute the area of the remaining cell and record it
        if it is less than the minimum area found so far ;
        if there is no next height then terminate ;
        ht := next height ;
    until false ;
end ;
```

Figure 2.9. Heuristic 3

number of tracks in the single channel. This takes $O(n \log n_i)$ time where n_i is the number of nets in channel i. Thus the time needed for the inner **repeat** loop when $k = 2$ is $O(cn \log n)$ (for uniform terminal distribution it is $O(cn \log(n/c))$. During the binary search, the heights corresponding to channel widths that require height $> ht$ are recorded. The minimum of these heights yields the next value of ht.

When $k > 2$, all track combinations for the $k - 1$ channels are tried as in Section 2.3. Again, each composite cell is broken up into its basic cells. As different track combinations are tried, we record the minimum height $> ht$ that results from any track combination. This gives the next value of ht. The time for the inner **repeat** loop is $O((c/(k-1))n(n/k)^{k-1})$ (or $O((c/(k-1))n(n/c)^{k-1})$ when terminals are uniformly distributed).

In all our experiments, the outer repeat loop was iterated fewer than $(k-1)n$ times. To ensure that the number of iterations is $O(kn)$, one may adopt the following scheme. When the number of iterations first reaches $(k-1)n$, compute a set of at most n new heights by beginning with the current constraint graph. This uses the current assignment of number of tracks in each channel. Heuristic 2 is next used to reduce the total number of available tracks by one and determine the height needed to complete the routing with the reduced number of tracks. This process gives us at most n new heights $h_1 < h_2 < \ldots < h_p$. Heuristic 3 is now resumed with h_1 as the next height. Only two iterations are performed. Then Heuristic 3 is resumed with $max\{\ h_2, ht\ \}$ as the next height. Again two iterations of the outer **repeat** loop are done. Next the heuristic is resumed with $max\{\ h_3, ht\ \}$ as the next height. This continues until we have gone through p resumptions of the heuristic. With this scheme to limit the number of iterations, the complexity of Heuristic 3 becomes $O(cn^2 \log n)$ when $k = 2$ and $O((c/(k-1))kn^2(n/k)^{k-1}) = O(cn^2(n/k)^{k-1})$ when $k > 2$. For the case when the n terminals are uniformly distributed over the c cells, the complexity is $O(cn^2 \log(n/c))$ when $k = 2$ and $O((c/(k-1))kn(n/c)^{k-1}) = O(cn(n/c)^{k-1})$ when $k > 2$. One may verify that since Heuristic 3 tries the maximum useful height (i.e., the height needed when no routing tracks are available), it generates optimal solutions when $k = c$.

2.5 Experimental Results

We programmed our three heuristics as well as the heuristic Fang [8] in C and ran tests on a single KSR processor. Optimal solutions for instances with up to nine

cells were obtained using the corrected version of the exhaustive search algorithm of Lim [16]. Our test set consisted of instances that had a number of cells, c, equal to one of the numbers in the set $\{3, \ldots, 9, 10, 20, 50, 100\}$. For each value of c, there were twenty instances and the results were averaged over these instances. An instance with c cells had $c-1$ routing channels. The number, t, of terminals on either side of each routing channel was equal to c for $3 \leq c \leq 9$ and was 10 for the other values of c. In addition, when $c = 100$, we also had instances with 20 terminals on either side. In our experiments, we considered only single layer and two layer routing.

Table 2.1 gives the average percentage by which the area of the single layer solutions generated by each of the heuristics exceeded the area of the single layer optimal solution. As is evident, each of the heuristics proposed in this chapter gave noticeably better solutions than did Fang. This table is only for the cases $3 \leq c \leq 9$ as for $c > 9$ the optimal algorithm of Lim [16] required too much time to complete. When $k \geq c$, Heuristic 3 is guaranteed to generate an optimal solution. So, we did not run these cases.

In table 2.2, we have used the single layer solution produced by Fang as the benchmark against which the solutions obtained by our three heuristics are compared. This table gives the average percentage by which the area of the solutions produced by our heuristics is less than that of the solutions produced by Fang. Our solutions have area 9 to 18% less.

Table 2.3 compares the computing time requirements of the various algorithms for the case of one layer. The optimal algorithm is useful only for small values of c

Table 2.1. Error rate (%) over optimal, $l = 1$

cells	t	Fang	Heuristic 1	Heuristic 2	Heuristic3		
					$k = 2$	$k = 3$	$k = 4$
3	3	5.9	0.5	0.2	0	*	*
4	4	10.0	0.9	0.1	0	0	*
5	5	11.0	3.7	0.3	0.3	0.1	0.1
6	6	12.7	2.4	0.3	0.2	0.2	0.0
7	7	16.1	3.5	0.3	0.1	0.1	0.1
8	8	17.9	2.7	0.4	0.3	0.3	0.1
9	9	18.8	3.5	0.3	0.4	0.3	0.1

t = number of terminals on each side of each routing channel

$*$: $k \geq c$

Table 2.2. Improvement (%) over Fang, $l = 1$

cells	t	Heuristic 1	Heuristic 2	Heuristic3		
				$k = 2$	$k = 3$	$k = 4$
10	10	9.4	14.0	14.0	14.2	14.3
20	10	10.5	15.4	15.4	15.6	15.6
50	10	10.6	16.1	16.1	16.2	–
100	10	9.1	16.5	16.5	16.6	–
100	20	9.3	18.3	18.0	–	–

– : excessive run time

Table 2.3. Time taken, $l = 1$

cells	t	Fang	Heuristic 1	Heuristic 2	Heuristic3			Optimal
					$k = 2$	$k = 3$	$k = 4$	
3	3	0.0	0.01	0.02	0.02	*	*	0.01
4	4	0.0	0.02	0.02	0.04	0.09	*	0.05
5	5	0.0	0.03	0.04	0.09	0.36	1.2	0.77
6	6	0.0	0.03	0.08	0.21	0.66	3.7	13
7	7	0.0	0.04	0.14	0.50	3.4	27	278
8	8	0.0	0.07	0.24	0.84	4.4	34	1.8†
9	9	0.0	0.09	0.42	1.8	16	72.3	47†
10	10	0.0	0.14	0.78	3.1	19	334	–
20	10	0.01	0.32	5.8	16	117	1196	–
50	10	0.01	1.4	93	127	896	–	–
100	10	0.03	4.4	782	590	4873	–	–
100	20	0.06	11	3022	4087	–	–	–

Times are in seconds.
† : Times are in hours.

(say up to 7). While Fang is significantly faster than the heuristics proposed here, the quality of the solutions generated by our heuristics is superior.

Table 2.4 is the analog of table 2.1 for the case of two layers. Again, our heuristics performed considerably better than did Fang. Table 2.5 gives the improvement in area due to increasing the number of routing layers from one to two. This is influenced somewhat by the width of cells which in our case ranged from 5 to 30 times the track separation. With narrower cells, the impact of the second layer would have been greater and with wider cells, it would have been less. Also, the impact of the second layer is more when more routing tracks are needed. For the smaller instances of table 2.4, for example, the optimal solutions with $l = 2$ required, on average, only 1.8% less area than when $l = 1$.

Table 2.4. Error rate (%) over optimal, $l = 2$

cells	t	Fang	Heuristic 1	Heuristic 2	Heuristic3		
					$k = 2$	$k = 3$	$k = 4$
3	3	6.3	0.5	0	0	*	*
4	4	9.8	1.1	0.1	0	0	*
5	5	11.0	2.7	0.1	0.2	0.1	0.0
6	6	13.0	1.4	0.1	0.0	0.0	0.1
7	7	15.9	2.3	0.1	0	0	0
8	8	16.8	1.7	0.2	0	0.1	0.0
9	9	18.4	2.2	0.1	0.1	0.1	0.0

Table 2.5. Improvement (%) over $l = 1$ cases

cells	t	Fang	Heuristic 1	Heuristic 2	Heuristic3		
					$k = 2$	$k = 3$	$k = 4$
10	10	4.1	4.7	3.3	3.3	3.0	3.0
20	10	4.1	5.8	3.4	3.4	3.3	3.2
50	10	4.6	5.2	3.4	3.2	3.2	–
100	10	4.7	6.3	3.5	3.4	3.4	–
100	20	7.1	10.4	4.8	5.2	–	–

Table 2.6. Improvement (%) over Fang, $l = 2$

cells	t	Heuristic 1	Heuristic 2	Heuristic3		
				$k = 2$	$k = 3$	$k = 4$
10	10	10.0	13.3	13.2	13.2	13.3
20	10	12.2	14.8	14.8	14.9	14.8
50	10	11.2	15.1	14.9	15.0	–
100	10	10.7	15.4	15.4	15.4	–
100	20	12.5	16.3	16.3	–	–

Table 2.6 is the analog of table 2.2 for the case of two layers. The results are similar to those in table 2.2. Table 2.7 gives the average computing times for the two layer instances. These are less than for the one layer case as the constraint graph has fewer edges.

For large c, we recommend the use of heuristic 2 or 3 (with $k = 2$) and for small c we recommend using heuristic 3 (with $k = 3$ or 4).

2.6 Conclusion

We have considered the problem of joining a row of compacted cells and developed heuristics to stretch cells and river-route the nets so that the layout area is minimized. Our proposed heuristic was compared, experimentally, with Fang [8] and found to produce layouts with less area. However, Fang is faster. We recommend the use of our Heuristic 3 with $k = 3$ or 4 in practice.

Table 2.7. Time taken, $l = 2$

cells	t	Fang	Heuristic 1	Heuristic 2	Heuristic3			Optimal
					$k = 2$	$k = 3$	$k = 4$	
3	3	0.0	0.01	0.02	0.02	*	*	0.0
4	4	0.0	0.02	0.02	0.03	0.06	*	0.04
5	5	0.0	0.02	0.02	0.07	0.25	0.75	0.60
6	6	0.0	0.02	0.04	0.13	0.38	2.0	11
7	7	0.0	0.04	0.06	0.29	1.9	14	223
8	8	0.0	0.06	0.12	0.53	2.5	19	$1.5^{\dagger}$
9	9	0.0	0.06	0.20	0.96	7.5	34	$39^{\dagger}$
10	10	0.01	0.09	0.32	1.7	9.9	153	–
20	10	0.01	0.24	2.7	9.9	65	651	–
50	10	0.01	1.1	42	85	586	–	–
100	10	0.02	3.7	357	472	3069	–	–
100	20	0.05	7.9	1377	3166	–	–	–

Times are in seconds.
† : Times are in hours.

CHAPTER 3
A NEW WEIGHT BALANCED BINARY SEARCH TREE

3.1 Introduction

A dictionary is a set of elements on which the operations of search, insert, and delete are performed. Many data structures have been proposed for the efficient representation of a dictionary [14]. These include direct addressing schemes such as hash tables and comparison schemes such as binary search trees, AVL-trees, red/black trees [12], trees of bounded balance [21], treaps [1], deterministic skip lists [20], and skip lists [26]. Of these schemes, AVL-trees, red/black trees, and trees of bounded balance (WB(α)) are balanced binary search trees. When representing a dictionary with n elements, using one of these schemes, the corresponding binary search tree has height $O(\log n)$ and individual search, insert, and delete operations take $O(\log n)$ time. When (unbalanced) binary search trees, treaps, or skip lists are used, each operation has an expected complexity of $O(\log n)$ but the worst case complexity is $O(n)$. When hash tables are used, the expected complexity is $O(1)$ per operation. However, the worst case complexity is $O(n)$. So, in applications where a worst case complexity guarantee is critical, one of the balanced binary search tree schemes is to be performed.

In this chapter, we develop a new balanced binary search tree called β-BBST (β-balanced binary search tree). Like WB(α) trees, this achieves balancing by controlling the relative number of nodes in each subtree. However, unlike WB(α) trees, during insert and delete operations, rotations are performed along the search path whenever they reduce the internal path length of the tree (rather than only when a subtree is out of balance). As a result, the constructed trees are expected to have a smaller internal path length than the corresponding WB(α) tree. Since the average search time is closely related to the internal path length, the time need to search in a β-BBST is expected to be less than that in a WB(α) tree.

In Section 3.2, we define the total search cost of a binary search tree and show that the rebalancing rotations performed in AVL and red/black trees might increase this metric. We also show that while similar rotations in WB(α) trees do not increase this metric, insert and delete operations in WB(α) trees do not avail of all opportunities to reduce the metric. In Section 3.3, we define β-BBSTs and show their relationship to WB(α) trees. Search, insert, and delete algorithms for β-BBSTs are developed in Section 3.4. A simplified version of β-BBSTs is developed in Section 3.5. Search, insert and delete operations for this version also take $O(\log n)$ time each. An even simpler version of β-BBSTs is developed in Section 3.6. For this version, we show that the average cost of an insert and search operation is $O(\log n)$ provided no deletes are performed.

An experimental evaluation of β-BBSTs and competing schemes for dictionaries (AVL, red/black, skip lists, etc.) was done and the results of this are presented in

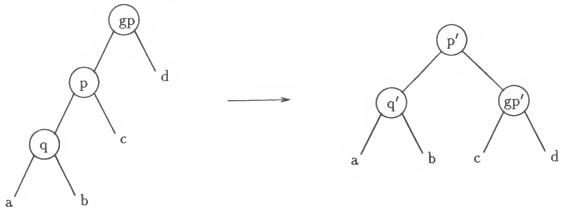
Section 3.7. This section also compares the relative performance of β-BBSTs and the two simplified versions of Sections 3.5 and 3.6.

3.2 Balanced Trees and Rotations

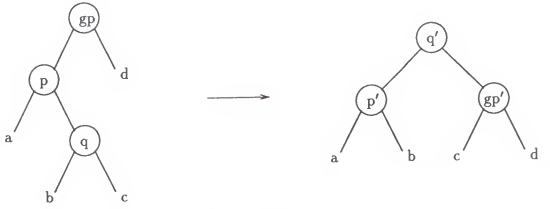
Following an insert or delete operation in a balanced binary search tree (e.g., AVL, red/black, WB(α), etc.), it may be necessary to perform rotations to restore balance. The rotations are classified as LL, RR, LR, and RL [14]. LL and RR rotations as well as LR and RL rotations are symmetric. While the conditions under which the rotations are performed vary with the class of balanced tree considered, the node movement patterns are the same. Figure 3.1 shows the transformation performed by an LL and an LR rotation. In this figure, nodes whose subtrees have changed as a result of the rotation are designated by a prime. So, p' is the original node p however its subtrees are different.

Let $h(x)$ be the height of the subtree with root x. Let $s(x)$ be the number of nodes in this subtree. When searching for an element x, x is compared with one element at each of $l(x)$ levels, where $l(x)$ is the level at which x is present (the root is at level 1). So, one measure of the "goodness" of the binary search tree, T, for search operations (assuming each element is searched for with equal probability) is its total search cost defined as:

$$C(T) = \sum_{x \in T} l(x).$$



(a) LL rotation



(b) LR rotation

Figure 3.1. LL and RL rotations

Notice that $C(T) = I(T) + n$ where $I(T)$ is the internal path length of T and n is the number of elements/nodes in T. The cost of unsuccessful searches is equal to the external path length $E(T)$. Since $E(T) = I(T) + 2n$, minimizing $C(T)$ also minimizes $E(T)$.

Total search cost is important as this is the dominant operation in a dictionary (note that insert can be modeled as an unsuccessful search followed by the insertion of a node at the point where the search terminated and deletion can be modeled by

a successful search followed by a physical deletion; both operations are then followed by a rebalancing/restructuring step).

Observe that in an actual implementation of the search operation in programming languages such as C++, C, and Pascal, the search for an x at level $l(x)$ will involve upto two comparisons at levels $1, 2, \ldots, l(x)$. If the code first checks $x = e_i$ where e_i is the element at level i to be compared and then $x < e_i$ to decide whether to move to the left or right subtree, then the number of element comparisons is exactly $2l(x) - 1$. In this case, the total number of element comparisons is

$$NC(T) = 2 \sum_{x \in T} l(x) - n = 2C(T) - n$$

and minimizing $C(T)$ also minimizes $NC(T)$. If the code first checks $x < e_i$ and then $x = e_i$ (or $> e_i$), the number of element comparisons done to find x is $l(x) + r(x) + 1$ where $r(x)$ is the number of right branches on the path from the root to x. The total number of comparisons is bounded by $2C(T)$. For simplicity, we use $C(T)$ to motivate our data structure.

In an AVL tree, when an LL rotation is performed, $h(q) = h(c)+1 = h(d)+1$ (see Figure 3.1(a)). At this time, the balance factor at gp is $h(p) - h(d) = 2$. The rotation restores height balance which is necessary to guarantee $O(\log n)$ search, insert, delete operations in an n node AVL tree. The rotation may, however, increase the total search cost. To see this, notice that an LL rotation affects the level numbers of only those nodes that are in the subtree with root gp prior to the rotation. We see that $l(q') = l(q) - 1, l(p') = l(p) - 1, l(gp') = l(gp) + 1$, the total search cost of the subtree

with root a is decreased by $s(a)$ as a result of the rotation, etc. Hence, the increase in $C(T)$ due to the rotation is:

$$l(p') - l(p) + l(q') - l(q) + l(gp') - l(gp) - s(a) - s(b) + s(d)$$

$$= -1 - 1 + 1 - s(q) + 1 + s(d) = s(d) - s(q).$$

A similar analysis shows that an LR rotation increases $C(T)$ by $s(d) - s(q)$.

If the LL rotation was triggered by an insertion, $s(q)$ is at least one more than the minimum number of nodes in an AVL tree of height $t = h(q) - 1$. So, $s(q) \geq \phi^{t+2}/\sqrt{5}$ where $\phi = (1 + \sqrt{5})/2$. The maximum value for $s(d)$ is $2^t - 1$. So, an LL rotation has the potential of increasing total search cost by as much as

$$2^t - 1 - \phi^{t+2}/\sqrt{5} \approx 2^t - 1 - 1.62^{t+2}/2.24.$$

This is negative for $t \leq 2$ and positive for $t > 2$. When $t = 10$, for example, an LL rotation may increase total search cost by as much as 877. As t gets larger, the potential increase in search cost gets much greater. This analysis is easily extended to the remaining rotations and also to red/black trees.

Definition (WB(α) [21]) The balance, $B(p)$, of a node p in a binary tree is the ratio $(s(l) + 1)/(s(p) + 1)$ where l is the left child of p. For $\alpha \in [0, 1/2]$, a binary tree T is in WB(α) iff $\alpha \leq B(p) \leq 1 - \alpha$ for every node p in T. By definition, the empty tree is in WB(α) for all α.

Lemma 1 (1) The maximum height, hmax(n), of an n node tree in WB(α) is $\sim \log_{\frac{1}{1-\alpha}}(n+1)$ *[21]*

(2) Inserts and deletes can be performed in an n node tree in WB(α) in O(log n) time for $2/11 < \alpha \leq 1 - \sqrt{2}/2$ *[4].*

(3) Each search operation in an n node tree in WB(α) takes O(log n) time [21].

In the case of weight balanced trees WB(α), an LL rotation is performed when $B(gp) \approx 1-\alpha$ and $B(p) \geq \alpha/(1-\alpha)$ (see Figure 3.1(a)) [21]. So,

$$1-\alpha \approx \frac{s(p)+1}{s(gp)+1} = \frac{s(p)+1}{s(p)+s(d)+2}$$

or

$$s(d) \approx s(p)\frac{\alpha}{1-\alpha} + \frac{2\alpha-1}{1-\alpha}$$

and

$$\frac{\alpha}{1-\alpha} \leq B(p) = \frac{s(q)+1}{s(p)+1}$$

or

$$s(q) \geq s(p)\frac{\alpha}{1-\alpha} + \frac{2\alpha-1}{1-\alpha}.$$

So, LL rotations (and also RR) do not increase the search cost. For LR rotations [21], $B(gp) \approx 1-\alpha$ and $B(p) < \alpha/(1-\alpha)$. So, $s(d) \approx s(p)\frac{\alpha}{1-\alpha} + \frac{2\alpha-1}{1-\alpha}$ and with respect to Figure 3.1(b),

$$\frac{\alpha}{1-\alpha} > B(p) = \frac{s(p)-s(q)}{s(p)+1}$$

or

$$s(q) > s(p)\frac{1-2\alpha}{1-\alpha} - \frac{\alpha}{1-\alpha}.$$

For $\alpha \leq 1/3$, $s(q) \geq s(d)$ and LR (RL) rotations do not increase search cost. Thus, in the case of WB(α) trees, the rebalancing rotations do not increase search cost. This statement remains true if the conditions for LL and LR rotation are changed to those in Blum and Mehlhorn [4].

While rotations do not increase the search cost of WB(α) trees, these trees miss performing some rotations that would reduce search cost. For example, it is possible to have $\alpha < B(gp) < 1-\alpha$, $B(p) \geq \frac{\alpha}{1-\alpha}$, and $s(q) > s(d)$. Since $B(gp)$ isn't high enough, an LL rotation isn't performed. Yet, performing such a rotation would reduce search cost.

3.3 β-BBSTs

Definition A *cost optimized search tree* (COST) is a binary search tree whose search cost cannot be reduced by performing a single LL, RR, LR, or RL rotation.

Theorem 8 *If T is a COST with n nodes, its height is at most* $\log_\phi(\sqrt{5}(n+1)) - 2$.

Proof Let N_h be the minimum number of nodes in a COST of height h. Clearly, $N_0 = 0$ and $N_1 = 1$. Consider a COST Q of height $h \geq 2$ having the minimum number of nodes N_h. Q has one subtree R whose height is $h-1$ and another, S, whose height is $\leq h-1$. R must be a minimal COST of height $h-1$ and so has N_{h-1} nodes. R, in return, must have one subtree, U, of height $h-2$ and another, V, of height $\leq h-2$. Both U and V are COSTs as R is a COST. Since R is a minimal

COST, U is a minimal COST of height $h-2$ and so has N_{h-2} nodes. Since Q is a COST, $|S| \geq max\{|U|, |V|\}$. We may assume that N_h is a nondecreasing function of h. So, $|S| \geq N_{h-2}$. Since Q is a minimal COST of height h, $|S| = N_{h-2}$. So,

$$N_h = N_{h-1} + N_{h-2} + 1, \ h \geq 2$$

$$N_0 = 0, N_1 = 1.$$

This recurrence is the same as that for the minimum number of nodes in an AVL tree of height h. So, $N_h = F_{h+2} - 1$ where F_i is the i'th Fibbonacci number. Consequently, $N_h \approx \phi^{h+2}/\sqrt{5} - 1$ and $h \leq \log_\phi(\sqrt{5}(n+1)) - 2$. □

Corollary 1 *The maximum height of a COST with n nodes is the same as that of an AVL tree with this many nodes.*

Definition Let a and b be the root of two binary trees. a and b are β-balanced, $0 \leq \beta \leq 1$, with respect to one another, denoted β-(a, b), iff

(a) $\beta(s(a) - 1) \leq s(b)$

(b) $\beta(s(b) - 1) \leq s(a)$

A binary tree T is β-balanced iff the children of every node in T are β-balanced.

A full binary tree is 1-balanced and a binary tree whose height equals its size (i.e., number of nodes) is 0-balanced.

Lemma 2 *If the binary tree T is β-balanced, then it is γ-balanced for $0 \leq \gamma \leq \beta$.*

Proof Follows from the definition of balance. □

Lemma 3 *If the binary tree T is β-balanced, $0 \leq \beta \leq 1/2$, then it is in WB(α) for $\alpha = \beta/(1+\beta)$.*

Proof Consider any node p in T. Let l and r be node p's left and right children.

$$B(p) = \frac{s(l)+1}{s(l)+s(r)+2} = \frac{1}{1+\frac{s(r)+1}{s(l)+1}}.$$

Since T is β-balanced, $s(l)-1 \leq s(r)/\beta$ or $s(l)+1 \leq s(r)/\beta+2$. So,

$$\frac{s(l)+1}{s(r)+1} \leq 1/\beta + \frac{2\beta-1}{\beta(s(r)+1)} \leq 1/\beta$$

or

$$\frac{s(r)+1}{s(l)+1} \geq \beta.$$

So, $B(p) \leq 1/(1+\beta)$. Further, $s(r)-1 \leq s(l)/\beta$. So,

$$\frac{s(r)+1}{s(l)+1} \leq 1/\beta.$$

And, $B(p) \geq 1/(1+1/\beta) = \beta/(1+\beta)$. Hence $\beta/(1+\beta) \leq B(p) \leq 1/(1+\beta)$ for every p in T. So, T is in WB(α) for $\alpha = \beta/(1+\beta)$. □

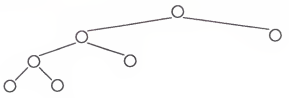


Figure 3.2. A tree in WB(1/4) that is not $\frac{1}{3}$-balanced

Remark While every β-balanced tree, $0 \leq \beta \leq 1/2$, is in WB(α) for $\alpha = \beta/(1+\beta)$, there are trees in WB(α) that are not β-balanced. Figure 3.2 shows an example of a tree in WB(1/4) that is not $\frac{1}{3}$-balanced.

Lemma 4 If T is a COST then T is $\frac{1}{2}$-balanced.

Proof If T is a COST, then every subtree of T is a COST. Consider any subtree with root p, left child l, and right child r. If neither l nor r exist, then $s(l) = s(r) = 0$ and p is $\frac{1}{2}$-balanced. If $s(l) = 0$ and $s(r) > 1$, then r has a nonempty subtree with root t and $s(t) > s(l)$. So p is not a COST. Hence, $s(r) \leq 1$ and p is $\frac{1}{2}$-balanced. The same is true when $s(r) = 0$. So, assume $s(l) > 0$ and $s(r) > 0$.

If $s(l) = 1$, then $s(r) \leq 3$ as otherwise, one of the subtrees of r has $m \geq 2$ nodes and $m > s(l)$ implies p is not a COST. Since $s(r) \leq 3$, $\frac{1}{2}(s(r)-1) \leq s(l)$ and $\frac{1}{2}(s(l)-1) \leq s(r)$. So, p is $\frac{1}{2}$-balanced. The same proof applies when $s(r) = 1$. When $s(l) > 1$ and $s(r) > 1$, let a and b be the roots of the left and right subtrees of l. Since p is a COST, $s(a) \leq s(r)$ and $s(b) \leq s(r)$. So, $s(l) = s(a) + s(b) + 1 \leq 2s(r) + 1$ and $\frac{1}{2}(s(l)-1) \leq s(r)$. Similarly, $\frac{1}{2}(s(r)-1) \leq s(l)$. So, $\frac{1}{2}$-(l, r). Since this proof applies to every nodes in T, the children of every p are $\frac{1}{2}$-balanced and T is $\frac{1}{2}$-balanced. □

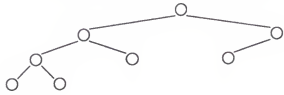


Figure 3.3. $\frac{1}{2}$-balanced tree that is not a COST

Remark There are $\frac{1}{2}$-balanced trees that are not COSTs (see Figure 3.3).

While a COST is in WB(1/3) and WB(α) trees can be maintained efficiently only for $2/11 < \alpha \leq 1 - 1/\sqrt{2} \approx 0.293$, a COST is better balanced than WB(α) trees with α in the usable range. Unfortunately, we are unable to develop $O(\log n)$ insert/delete algorithms for a COST.

In the next section, we develop insert and delete algorithms for β-balanced binary search trees (β-BBST) for $0 < \beta \leq \sqrt{2} - 1$. Note that every $(\sqrt{2} - 1)$-BBST is in WB(α) for $\alpha = 1 - 1/\sqrt{2}$ which is the largest permissible α. Since our insert and delete algorithms perform rotations along the search path whenever these result in improved search cost, BBSTs are expected to have better search performance than WB(α) trees (for $\alpha = \beta/(1 + \beta)$).

Each node of a β-BBST has the fields LeftChild, Size, Data, and RightChild. Since every β-BBST, $\beta > 0$, is in WB(α), for $\alpha > 0$, β-BBSTs have height that is logarithmic in n, the number of nodes (provided $\beta > 0$).

3.4 Search, Insert, and Delete in a β-BBST

To reduce notational clutter, in the rest of the chapter, we abbreviate $s(a)$ by a (i.e., the node name denotes subtree size).

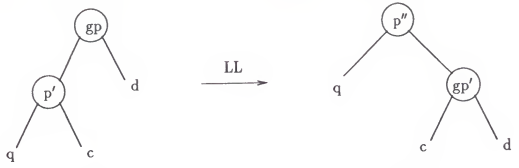
3.4.1 Search

This is done exactly as in any binary search tree. Its complexity is $O(h)$ where h is the height of the tree. Notice that since each node has a size field, it is easy to perform a search based on index (i.e., find the 10'th smallest key). Similarly, our insert and delete algorithms can be adapted to indexed insert and delete.

3.4.2 Insertion

To insert a new element x into a β-BBST, we first search for x in the β-BBST. This search is unsuccessful (as x is not in the tree) and terminates by falling off the tree. A new node y containing x is inserted at the point where the search falls off the tree. Let p' be the parent (if any) of the newly inserted node. We now retrace the path from p' to the root performing rebalancing rotations.

There are four kinds of rotations LL, LR, RL, and RR. LL and RR rotations are symmetric and so also are LR and RL rotations. The typical configuration before an LL rotation is performed is given in Figure 3.4(a). p' denotes the root of a subtree in which the insertion was made. Let p be the (size of the) subtree before the insertion. Then, since the tree was a β-BBST prior to the insertion, β-(p, d). Also, for the LL rotation to be performed, we require that $(q \geq c)$ and $(q > d)$. Note that $q > d$ implies $q \geq 1$. We shall see that β-(q, c) follows from the fact that the insertion is



(a) before (b) after

Figure 3.4. LL rotation for insertion

made into a β-BBST and from properties of the rotation. Following an LL rotation, p' is updated to be the node p''.

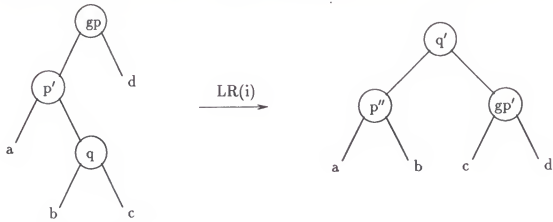
Lemma 5 [LL insertion lemma] If [β-$(p,d) \wedge \beta$-$(q,c) \wedge (q \geq c) \wedge (q > d)$] for $0 \leq \beta \leq 1/2$ before the rotation, then β-(q, gp') and β-(c,d) after the rotation.

Proof Assume the before condition.

(a) $\beta(q-1) \leq c$ (as β-(q,c)) $< gp'$. Also, $\beta(gp'-1) = \beta(c+d) \leq 2\beta q$ (as $\beta \geq 0, q \geq c$ and $q > d$) $\leq q$ (as $\beta \leq 1/2$). So, β-(q, gp').

(b) $d < q \Rightarrow d - 1 < q - 1 \Rightarrow \beta(d-1) \leq \beta(q-1) \leq c$ (as β-(q,c)). Also, $\beta(c-1) \leq \beta(q+c-1) = \beta(p'-2) = \beta(p-1) \leq d$ (as β-(p,d)). So, β-(c,d). □

In an LR rotation, the before configuration is as in Figure 3.4(a). However, this time $q < c$. Figure 3.4(a) is redrawn in Figure 3.5(a). In this, the node labeled c in Figure 3.4(a) has been labeled q and that labeled q in Figure 3.4(a) has been labeled



(a) before (b) after substep (i)

Figure 3.5. Substep (i) of insertion LR rotation

a. With respect to the labelings of Figure 3.5(a), rotation LR is applied when

$$[(q > a) \wedge (q > d)].$$

The other conditions that apply when an LR rotation is performed are

$$[\beta\text{-}(p,d) \wedge \beta\text{-}(a,q) \wedge \beta\text{-}(b,c)].$$

Here p denotes the (size of the) left subtree of gp prior to the insertion. An LR rotation is accomplished in two substeps (or two subrotations). The first of these is shown in Figure 3.5(b). Following an LR rotation, p' is updated to be node q'.

Lemma 6 [LR substep(i) insertion lemma] If $[\beta\text{-}(p,d) \wedge \beta\text{-}(a,q) \wedge \beta\text{-}(b,c) \wedge (q > a) \wedge (q > d)]$ *for* $0 \leq \beta \leq 1/2$ *before the subrotation, then* $[\beta\text{-}(p'', gp') \wedge \{(\beta\text{-}(a,b) \wedge \frac{\beta}{1+\beta}\text{-}(c,d)) \vee (\frac{\beta}{1+\beta}\text{-}(a,b) \wedge \beta\text{-}(c,d))\}]$ *after the subrotation.*

Proof Assume the before condition. First, we show that β-(p'', gp') after the rotation. Note that $\beta(p''-1) = \beta(a+b) = \beta(a+b+c+1) - \beta(c+1) = \beta(p'-1) - \beta(c+1) = \beta(p-1) - \beta c \leq d - \beta c \leq d < gp'$. Also, $\beta(gp'-1) = \beta(c+d) \leq b+\beta+\beta d$ (as β-(b,c)) $\leq b+\beta q$ (as $q > d$) $\leq b+a+\beta$ (as β-(a,q)) $< p''$ (as $\beta \leq 1/2$ and $p'' = a+b+1$). So, β-(p'', gp').

Next, we prove two properties that will be used to complete the proof.

P1: $\beta(b-1) \leq a$.

To see this, note that $\beta(b-1) \leq \beta(q-1) \leq a$ (as β-(a,q)).

P2: $\beta(c-1) \leq d$.

For this, observe that $p'-1 = a+q \geq \beta(q-1)+q$ (as β-(a,q)) $= (\beta+1)(q-1)+1$. So, $q-1 \leq \frac{p'-2}{\beta+1} = \frac{p-1}{\beta+1}$. Similarly, $q-1 = b+c \geq \beta(c-1)+c$ (as β-(b,c)) $= (\beta+1)(c-1)+1$. So, $\beta(c-1) \leq \frac{\beta}{\beta+1}(q-2) \leq \frac{\beta}{\beta+1}(q-1) \leq \frac{\beta(p-1)}{(\beta+1)^2} \leq \frac{d}{(\beta+1)^2}$ (as β-(p,d)) $\leq d$.

To complete the proof of the lemma, we need to show

$$\{(\beta\text{-}(a,b) \wedge \frac{\beta}{1+\beta}\text{-}(c,d)) \vee (\frac{\beta}{1+\beta}\text{-}(a,b) \wedge \beta\text{-}(c,d))\}.$$

We do this by considering the two cases $b \geq c$ and $b < c$.

Case $b \geq c$: Since $a < q = b+c+1, \beta(a-1) \leq \beta(b+c) \leq 2\beta b \leq b$. This and P1 imply β-(a,b). Also, $d < q = b+c+1$. So, $\frac{\beta}{\beta+1}(d-1) \leq \frac{\beta}{\beta+1}(b+c-1) = \frac{\beta}{\beta+1}c + \frac{\beta}{\beta+1}(b-1) \leq \frac{\beta}{\beta+1}c + \frac{c}{\beta+1}$ (as β-(b,c)) $= c$. This, together with P2 implies $\frac{\beta}{1+\beta}$-(c,d). So, β-$(a,b) \wedge \frac{\beta}{1+\beta}$-$(c,d)$.

Case $b < c$: Since $a < q = b+c+1, a-1 < b+c$. So, $a-1 \leq b+c-1$ or

$\frac{\beta(a-1)}{1+\beta} \leq \frac{\beta b}{1+\beta} + \frac{\beta(c-1)}{1+\beta} \leq \frac{\beta b}{1+\beta} + \frac{b}{1+\beta}$ (as β-$(b,c)) = b$. This and P1 imply $\frac{\beta}{1+\beta}$-(a,b). Also, $d-1 \leq q-2 = b+c-1$. So, $\beta(d-1) \leq \beta(b+c-1) \leq \beta(2c-1) \leq c$. This, together with P2 implies β-(c,d). So, $\frac{\beta}{1+\beta}$-$(a,b) \wedge \beta$-(c,d). □

Since an LR(i) rotation can cause the tree to lose its β-balance property, it is necessary to follow this with another rotation that restores the β-balance property. It suffices to consider the two cases of Figures 3.6 and 3.7 for this follow up rotation. The remaining cases are symmetric to these. In Figures 3.6 and 3.7, p and d denote the nodes that do not satisfy β-(p,d). Note, however, that these nodes do satisfy $\frac{\beta}{1+\beta}$-(p,d).

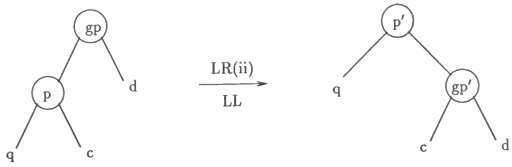
Since the follow up rotation to LR(i) is done only when

$$\frac{\beta}{1+\beta}\text{-}(p,d) \wedge (\neg\beta\text{-}(p,d)),$$

either $\beta(p-1) > d$ or $\beta(d-1) > p$. When $\beta(p-1) > d$, the second substep rotation is one of the two given in Figures 3.6 and 3.7. When $\beta(d-1) > p$, rotations symmetric to these are performed. In the following, we assume $\beta(p-1) > d$. Further, we may assume $d > 0$, as $d = 0$ and $\frac{\beta}{1+\beta}$-(p,d) imply $p \leq 1$. Hence, β-(p,d). Also, $d > 0$ and $\beta(p-1) > d$ imply $p > 1$.

The LR(ii) LL rotation is done when the condition

$$A = (q > d) \wedge (c < (1+\beta)q + (1-\beta)) \wedge B \quad \text{where}$$



(a) before (b) after

Figure 3.6. Case LL for LR(ii) rotation

$$B = \frac{\beta}{1+\beta}\text{-}(p,d) \wedge (\neg\beta\text{-}(p,d)) \wedge \beta\text{-}(q,c) \wedge (\beta(p-1) > d > 0).$$

Lemma 7 [Case LR(ii) LL rotation] If A holds before the rotation of Figure 3.6, then β-(q, gp') *and* β-(c, d) *after the rotation provided* $0 < \beta \leq \sqrt{2} - 1$.

Proof (a) β-(q, gp'):

$\beta(q-1) \leq c$ (as β-$(q,c)) < gp'$. Also, $\beta(gp'-1) = \beta(c+d) < \beta((1+\beta)q+(1-\beta)+d) \leq$ $\beta(1+\beta)q + \beta(1-\beta) + \beta(q-1)$ (as $q > d$) $= \beta(2+\beta)q - \beta^2 < q$ (as $\beta(2+\beta) \leq 1$ for $0 < \beta \leq \sqrt{2} - 1$). So, β-(q, gp').

(b) β-(c, d):

$\beta(d-1) < \beta(q-1) \leq c$ (as β-(q,c)). And, $\beta(c-1) = \frac{\beta^2}{1+\beta}(c-1) + \frac{\beta}{1+\beta}(c-1) \leq$ $\frac{\beta}{1+\beta}q + \frac{\beta}{1+\beta}(c-1) = \frac{\beta}{1+\beta}(q+c-1) = \frac{\beta}{1+\beta}(p-2) < \frac{\beta}{1+\beta}(p-1) \leq d$ (as $\frac{\beta}{1+\beta}$-(p,d)). So, β-(c, d). □

Lemma 8 If $(c < (1+\beta)q + (1-\beta)) \wedge (\beta(p-1) > d)$ *in Figure 3.6, then* $d \leq q$ *provided* $0 < \beta \leq \sqrt{2} - 1$.

Proof Since $d < \beta(p-1) = \beta(q+c) < \beta(q+(1+\beta)q+1-\beta) = \beta(\beta+2)q+\beta(1-\beta) < q+1$ (as $\beta(\beta+2) \leq 1$ and $\beta(1-\beta) < 1$ for $0 < \beta \leq \sqrt{2}-1$). So, $d \leq q$. □

So, the only time an LR(ii) LL rotation is not done is when $C = (C_1 \vee C_2) \wedge B$ holds where

$$C_1 = (q = d) \wedge (c < (1+\beta)q + 1 - \beta)$$

$$C_2 = c \geq (1+\beta)q + (1-\beta).$$

At this time, the LR rotation of Figure 3.7 is done. In terms of the notation of Figure 3.7, the condition C becomes $D = (D_1 \vee D_2) \wedge E$ where

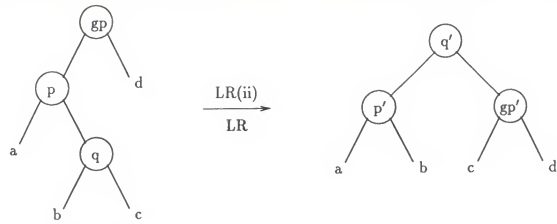
$$D_1 = (a = d) \wedge (q < (1+\beta)a + 1 - \beta)$$

$$D_2 = q \geq (1+\beta)a + 1 - \beta$$

$$E = \frac{\beta}{1+\beta}\text{-}(p,d) \wedge \neg\beta\text{-}(p,d) \wedge \beta\text{-}(a,q) \wedge \beta\text{-}(b,c) \wedge (\beta(p-1) > d > 0).$$

Lemma 9 When an LR(ii) LR rotation is performed and $\beta \leq \sqrt{2}-1$, $q > d$ and so search cost is reduced.

Proof If D_1, then since $d < \beta(p-1) = \beta(a+q) = \beta(d+q)$, $q > d/\beta - d > d$ as $\beta \leq \sqrt{2}-1$. If D_2, then $d < \beta(p-1) = \beta(a+q) \leq \beta(\frac{q-1+\beta}{1+\beta}+q) = \frac{\beta(2+\beta)}{1+\beta}q - \frac{\beta(1-\beta)}{1+\beta} < \frac{\beta(2+\beta)}{1+\beta}q \leq q$ (as $\beta \leq \sqrt{2}-1$). □



(a) before (b) after

Figure 3.7. Case LR for LR(ii) rotation

Lemma 10 *When* $(d = a) \wedge \beta\text{-}(b, c) \wedge (\beta(p-1) > d) \wedge (\beta \leq \sqrt{2}-1)$ *(see Figure 3.7),* $\beta(a-1) \leq b$ *and* $\beta(d-1) \leq c$.

Proof Since $\beta(p-1) > d$ and $d = a, \beta(p-1) > a$ or $\beta(a+q) > a$ or $a(1-\beta) < \beta q$ or $a < \frac{\beta}{1-\beta}q$. So, $\beta(a-1) < \frac{\beta^2}{1-\beta}q - \beta = \frac{\beta^2}{1-\beta}(b+c+1) - \beta$.

If $c \leq \frac{b}{\beta} + \beta$, then

$$\begin{aligned}
\beta(a-1) &< \frac{\beta^2}{1-\beta}(b + \frac{b}{\beta} + \beta + 1) - \beta \\
&= \frac{\beta(\beta+1)b}{1-\beta} + \frac{\beta^2(\beta+1)}{1-\beta} - \beta \\
&= \frac{\beta(\beta+1)b}{1-\beta} + \frac{\beta(\beta^2+\beta-1+\beta)}{1-\beta} \\
&= \frac{\beta(\beta+1)b}{1-\beta} + \frac{\beta(\beta^2+2\beta-1)}{1-\beta} \\
&\leq b \ \text{(as } \beta(\beta+1) \leq 1-\beta \text{ for } \beta \leq \sqrt{2}-1 \text{ and } \beta^2+2\beta-1 \leq 0 \text{ for } \beta \leq \sqrt{2}-1).
\end{aligned}$$

Since $\beta(c-1) \leq b, c \leq \frac{b}{\beta}+1$. So,

$$\beta(a-1) < \frac{\beta^2}{1-\beta}(b+c+1) \leq \frac{\beta^2}{1-\beta}(b+\frac{b}{\beta}+2) \leq \frac{\beta(\beta+1)b}{1-\beta} + \frac{3\beta^2-\beta}{1-\beta}.$$

So,

$$a-1 < \frac{\beta+1}{1-\beta}b + \frac{3\beta-1}{1-\beta}.$$

However, since $\beta^2+2\beta-1 \leq 0$ for $\beta \leq \sqrt{2}-1$, $(1+\beta)/(1-\beta) \leq \frac{1}{\beta}$ and $(3\beta-1)/(1-\beta) \leq \beta$. So, $a-1 < b/\beta+\beta$. If $a \geq c+1$, then $c \leq a-1 < b/\beta+\beta$. We have already shown that for $c \leq b/\beta+\beta$, $\beta(a-1) \leq b$. So, assume $a < c+1$. Now, $a \leq c$ and $\beta(a-1) \leq \beta(c-1) \leq b$ (as β-(b,c)). So, $\beta(a-1) \leq b$ in all cases. $\beta(a-1) \leq c$ may be shown in a similar way. Since $a = d$, we get $\beta(d-1) \leq c$. □

Lemma 11 [Case LR(ii) LR rotation] If D holds before the rotation of Figure 3.7, then β-(p', gp'), β-(a,b), and β-(c,d) following the rotation provided $0 < \beta \leq \sqrt{2}-1$.

Proof (a) β-(p', gp'):

$\beta(gp'-1) = \beta(c+d) \leq b+\beta+\beta d$ (as β-(b,c)) $\leq b+\beta+\beta q$ (from Lemmas 9 and 10, $q \geq d$) $\leq b+\beta+a+\beta = a+b+2\beta < a+b+1 = p'$. Also, since $\frac{\beta}{1+\beta}$-(p,d) and $q \geq d, \beta(p-1) \leq (\beta+1)d$ or $\beta(a+q) \leq (\beta+1)d$ or $a+q \leq (1+\frac{1}{\beta})d$ or $a \leq (1+\frac{1}{\beta})d-q \leq (1+\frac{1}{\beta})d-d = d/\beta$. So, $\beta(p'-1) = \beta(a+b) < d+\beta b \leq d+c+\beta$ (as β-(b,c)) $< d+c+1 = gp'$.

(b) β-(a,b):

Since $b \leq q$ and β-(a,q), $\beta(b-1) \leq \beta(q-1) \leq a$.

When D_1, $\beta(a-1) \leq b$ was proved in Lemma 10. So, β-(a,b).

When D_2, $q \geq a(1+\beta)+1-\beta$. So,

$$a \leq \frac{q}{1+\beta} - \frac{1-\beta}{1+\beta} = \frac{b+c+1}{1+\beta} - \frac{1-\beta}{1+\beta}.$$

So,

$$\beta(a-1) \leq \frac{\beta b+\beta c+\beta}{1+\beta} - \frac{1-\beta}{1+\beta}\beta - \beta \leq \frac{\beta b+b+2\beta}{1+\beta} - \frac{1-\beta}{1+\beta}\beta - \beta = b.$$

So, β-(a,b).

(c) β-(c,d):

Note that $\beta(c-1) < \beta(q-1) < \frac{\beta}{1+\beta}(q-1) < \frac{\beta}{1+\beta}(p-1) \leq d$.

When D_1, $\beta(d-1) \leq c$ was proved in Lemma 10. So, β-(c,d).

When D_2, if $d < b+1$, then $d \leq b$ and $\beta(d-1) \leq \beta(b-1) \leq c$. So, assume $d \geq b+1$.

Now, $b \leq d-1 < \beta(p-1)-1$. So,

$$\begin{aligned}
b &< \beta(a+b+c+1)-1 \\
&\leq \beta(\frac{q-1+\beta}{1+\beta}+b+c+1)-1 \\
&= \frac{\beta}{1+\beta}(b+c+\beta+(1+\beta)(b+c+1))-1 \\
&\leq \frac{\beta}{1+\beta}(\frac{c}{\beta}+1+c+\beta+(1+\beta)(\frac{c}{\beta}+1+c+1))-1 \\
&= \frac{\beta}{1+\beta}(\frac{\beta+1}{\beta}c+(1+\beta)+(1+\beta)(\frac{1+\beta}{\beta}c+2))-1 \\
&= c+\beta+(1+\beta)c+2\beta-1 \\
&= (2+\beta)c+3\beta-1 < (2+\beta)c+\beta \text{ (as } \beta \leq \sqrt{2}-1)
\end{aligned}$$

$$\leq \frac{c}{\beta} + \beta \text{ (as } \beta \leq \sqrt{2} - 1).$$

Also, from $d < \beta(p-1)$ and the above derivation, we get

$$\begin{aligned}
d &< \frac{\beta}{1+\beta}(b + c + \beta + (1+\beta)(b + c + 1)) \\
&\leq \frac{\beta}{1+\beta}(\frac{c}{\beta} + \beta + c + \beta + (1+\beta)(\frac{c}{\beta} + \beta + c + 1)) \\
&= \frac{\beta}{1+\beta}(\frac{\beta+1}{\beta}c) + \frac{2\beta^2}{1+\beta} + \beta(\frac{1+\beta}{\beta}c) + \beta(\beta+1) \\
&= (2+\beta)c + \frac{2\beta^2}{1+\beta} + \beta(\beta+1) \\
&= (2+\beta)c + \frac{2\beta^2 + \beta^2 + \beta^3 + \beta + \beta^2}{1+\beta} \\
&= (2+\beta)c + \frac{\beta^3 + 4\beta^2 + \beta}{1+\beta} \\
&\leq (2+\beta)c + 1 \text{ (as } \beta^3 + 4\beta^2 + \beta < 1 + \beta \text{ for } \beta \leq \sqrt{2} - 1).
\end{aligned}$$

So, $\beta(d-1) \leq \beta(2+\beta)c \leq c$ (as $\beta \leq \sqrt{2} - 1$). So, β-(c, d). □

Theorem 9 *If T is β-balanced, $0 \leq \beta \leq \sqrt{2} - 1$, prior to insertion, it is so following the insertion.*

Proof First note that since all binary search trees are balanced for $\beta = 0$, the rotations (while unnecessary) preserve 0-balance. So, assume $\beta > 0$. Consider the tree T' just after the new element has been inserted but before the backward restructuring pass begins.

If the newly inserted node, z, has no parent in T', then T was empty and T' is β-balanced. If z has a parent but no grandparent, then T has at most one nonempty

subtree X. Since T is β-balanced, $\beta(|X|-1) \leq 0$. So, $|X| \leq 1$. Following the insertion, T' has one subtree with ≤ 1 nodes and one with exactly one. So, T' is β-balanced. We may therefore assume that z has a grandparent in T'.

From the downward insertion path, it follows that all nodes u in T' that have children l and r for which $\neg\beta$-(l,r) must lie on the path from the root to z. During the backward restructuring pass, each node on this path (other than z and its parent) play the role of *gp* in Figures 3.4 and 3.5. The β-property cannot be violated at z as z has no children. It cannot be violated at the parent, s, of z as s satisfied the β-property prior to insertion. As a result its other subtree has ≤ 1 element. So, following the insertion, s satisfies the β-property. As a result, each node in T' that might possibly violate the β-property becomes the *gp* node during the restructuring pass. Consider one such *gp* node. It has children in T' denoted by p' and d. Its children in T are p and d. Figures 3.4 and 3.5 show the case when d is the right subtree of *gp* in both T and T'. The cases RR and RL arise when d is the left subtree.

During the restructuring pass, *gp* begins at the grandparent of z and moves up to the root of T'. If z is at level r in T', (the root being at level 1), then *gp* takes on $r-2$ values during the restructuring pass. We shall show that at each of these $r-2$ positions either

(a) no rotation is performed and all descendants of *gp* satisfy the β-property or

(b) a rotation is performed and following this, all descendants of node p'' (Figure 3.4) or of node q' (Figure 3.5) satisfy the β-property.

As a result, following the rotation (if any) performed when gp becomes the root of T', the restructured tree is β-balanced. The proof is by induction on r. When $r = 3$ (recall, we assume z has a grandparent), gp begins at the root of T' and its descendants satisfy the β-property.

Without loss of generality, assume that the insertion took place in the left subtree of gp. With respect to Figure 3.4, we have three cases: (i) $q \geq c$ and $q > d$, (ii) $q < c$ and $c > d$, and (iii) $q \leq d$ and $c \leq d$. In case (i), all conditions for an LL rotation hold and such a rotation is performed. In case (ii), an LR rotation is performed. Following either rotation, T' is β-balanced. In case (iii), $\beta(p' - 1) = \beta(q + c) \leq 2\beta d < d$ (as $\beta \leq \sqrt{2} - 1$). Also, $\beta(d - 1) \leq p < p + 1 = p'$. So, $\beta(d - 1) < p'$. Hence, β-(p', d) and T' is β-balanced.

For the induction hypothesis, assume (a) and (b) whenever $r \leq k$. In the induction step, we show (a) and (b) for trees T with $r = k + 1$. The subtree in which the insertion is done has $r = k$. So, (a) and (b) hold for all gp locations in the subtree. We need to show (a) and (b) only when gp is at the root of T'. This follows from Lemmas 5, 6, 7, and 11.

The theorem now follows. □

Lemma 12 The time needed to do an insertion in an n node β-BBST is $O(\log n)$ provided $0 < \beta \leq \sqrt{2} - 1$.

Proof Follows from the fact that insertion takes $O(h)$ time where h is the tree height and $h = O(\log n)$ when $\beta > 0$ (Lemmas 1 and 3). □

3.4.3 Deletion

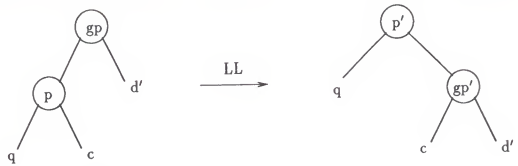
To delete element x from a β-BBST, we first use the unbalanced binary search tree deletion algorithm of Horowitz and Sahni [14] to delete x and then perform a series of rebalancing rotations. The steps are:

Step 1 [Locate x] Search the β-BBST for the node y that contains x. If there is no such node, terminate.

Step 2 [Delete x] If y is a leaf, set d' to nil, gp to the parent of y, and delete node y. If y has exactly one child, set d' to be this child; change the pointer from the parent (if any) of y to point to the child of y; delete node y; set gp to be the parent of d'. If y has two children, find the node z in the left subtree of y that has largest value; move this value into node y; set $y = z$; go to the start of Step 2. { note that the new y has either 0 or 1 child }

Step 3 [Rebalance] Retrace the path from d' to the root performing rebalancing rotations.

There are four rebalancing rotations LL, LR, RR, and RL. Since LL and RR as well as LR and RL are symmetric rotations, we describe LL and LR only. The discussion is very similar to the case of insertion. The differences in proofs are due to the fact that a deletion reduces the size of encountered subtrees by 1 while an



(a) before (b) after

Figure 3.8. LL rotation for deletion

insertion increases it by 1. In an LL rotation, the configuration just before and after the rotation is shown in Figure 3.8. This rotation is performed when $q \geq c$ and $q > d'$. Following the rotation, d' is updated to the node p'.

Let d denote the size of the right subtree of gp before the deletion. So, $d = d' + 1$. Since prior to the deletion the β-BBST was β-balanced, it follows that β-(p, d) and β-(q, c).

Lemma 13 [LL deletion lemma] If [β-$(p, d) \wedge \beta$-$(q, c) \wedge (q \geq c) \wedge (q > d) \wedge (1/3 \leq \beta \leq 1/2)$] before the rotation, then [β-$(q, gp') \wedge \beta$-(c, d')] after the rotation.

Proof (a) β-(q, gp'):

$\beta(q-1) \leq c$ (as β-(q, c)) $< gp'$. Also, $\beta(gp' - 1) = \beta(c + d') < 2\beta q$ (as $c \leq q$ and $d' < q$) $\leq q$ (as $\beta \leq 1/2$). So, β-(q, gp').

(b) β-(c, d'):

$d' < q \Rightarrow d' - 1 < q - 1 \Rightarrow \beta(d' - 1) < \beta(q - 1) \leq c$. Also, when $c \leq 1$, $\beta(c-1) \leq 0 \leq d'$ (as $d' \geq 0$). When $c > 1, q \geq c \Rightarrow q \geq 2$ and $p = q + c + 1 \geq c + 3$. So, $\beta(c-1) \leq \beta(p-1) - 3\beta \leq d - 3\beta$ (as β-(p, d)) $\leq d - 1$ (as $\beta \geq 1/3$) $= d'$. Hence,

β-(c, d'). □

In an LR rotation, the before configuration is as in Figure 3.8(a). However, this time $q < c$. Figure 3.8(a) is redrawn in Figure 3.9(a). In this, the node labeled c in Figure 3.8(a) has been relabeled q and that labeled q in Figure 3.8(a) has been relabeled a. With respect to the labelings of Figure 3.9(a), rotation LR is applied when

$$[(q > a) \wedge (q > d')].$$

The other conditions that apply when an LR rotation is performed are

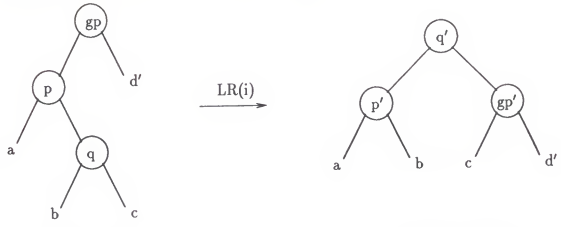
$$[\beta\text{-}(p, d) \wedge \beta\text{-}(a, q) \wedge \beta\text{-}(b, c)].$$

Here d denotes the (size of) right subtree of gp prior to the deletion. As in the case of insertion, an LR rotation is accomplished in two substeps (or two subrotations). The first of these is shown in Figure 3.9. Following an LR rotation, d' is updated to node q'.

Lemma 14 [LR substep(i) deletion lemma] If $[\beta\text{-}(p, d) \wedge \beta\text{-}(a, q) \wedge \beta\text{-}(b, c) \wedge (q > a) \wedge (q > d')]$ *before the subrotation LR(i), then* $[\beta\text{-}(p', gp') \wedge \{(\beta\text{-}(a, b) \wedge \frac{\beta}{1+\beta}\text{-}(c, d')) \vee (\frac{\beta}{1+\beta}\text{-}(a, b) \wedge \beta\text{-}(c, d'))\}]$ *after the subrotation provided* $1/3 \leq \beta \leq 1/2$.

Proof Assume the before condition.

(a) If $b = c = 0$, then $q = b + c + 1 = 1$. Furthermore, $(q > a)$ and $(q > d')$ imply $a = d' = 0$. So, $gp' = p' = 1$. Hence, $[\frac{1}{2}\text{-}(p', gp') \wedge \frac{1}{2}\text{-}(a, b) \wedge \frac{1}{2}\text{-}(c, d')]$



(a) before (b) after substep (i)

Figure 3.9. LR rotation for deletion

(b) If $b = 1$ and $c = 0$, then $q = 2, a \leq 1$, and $d' \leq 1$. So, $1 \leq p' \leq 3$ and $1 \leq gp' \leq 2$. Hence, $[\frac{1}{2}\text{-}(p', gp') \wedge \frac{1}{2}\text{-}(a, b) \wedge \frac{1}{2}\text{-}(c, d')]$

(c) If $b = 0$ and $c = 1$, then $q = 2, a \leq 1$, and $d' \leq 1$. So, $1 \leq p' \leq 2$ and $1 \leq gp' \leq 3$. Hence, $[\frac{1}{2}\text{-}(p', gp') \wedge \frac{1}{2}\text{-}(a, b) \wedge \frac{1}{2}\text{-}(c, d')]$

As a result of (a) – (c), to complete the proof, we may assume that $b \geq 1$ and $c \geq 1$. So, $q \geq 3, a \geq 1$ (as $\beta\text{-}(a, q) \Rightarrow \beta(q-1) \leq a$ or $a \geq 2\beta > 0$), $p = a + q + 1 \geq 5, d \geq 2$ (as $\beta\text{-}(p, d) \Rightarrow \beta(p-1) \leq d$ and $\beta \geq 1/3$), and $d' = d - 1 \geq 1$.

First, we show that $\beta\text{-}(p', gp')$. For this, note that $a + b + c + 1 = p - 1$. From $\beta\text{-}(p, d)$, it follows that $\beta(a + b + c + 1) = \beta(p - 1) \leq d$. So, $\beta(a + b) \leq d - \beta c - \beta$. From Figure 3.9(b), we see that $\beta(p' - 1) = \beta(a + b)$. Hence, $\beta(p' - 1) \leq d - \beta c - \beta = d' - \beta c + 1 - \beta \leq d' + 1 - 2\beta < gp'$. Also,

$$\begin{aligned} \beta(gp' - 1) = \beta(c + d') &\leq b + \beta + \beta d' \text{ (as } \beta\text{-}(b, c)) \\ &< b + \beta q + \beta \text{ (as } q > d') \end{aligned}$$

$$\leq \; b + a + 2\beta \text{ (as } \beta\text{-}(a,q))$$

$$< \; p'.$$

So, β-(p', gp').

Next, we prove two properties that will be used to complete the proof.

P1: $\beta(b-1) \leq a$.

To see this, note that $\beta(b-1) < \beta(q-1) \leq a$ (as β-(a,q)).

P2: $\beta(c-1) \leq d'$.

For this, observe that $\beta(c-1) \leq \beta(q-2)$ (as $c \leq q-1$) $\leq \beta(p-4)$ (as $q = p-a-1$ and $a \geq 1$) $= \beta(p-1) - 3\beta \leq d-1$ (as β-(p,d) and $\beta \geq 1/3$) $= d'$.

To complete the proof of the lemma, we need to show

$$\{(\beta\text{-}(a,b) \wedge \frac{\beta}{1+\beta}\text{-}(c,d')) \vee (\frac{\beta}{1+\beta}\text{-}(a,b) \wedge \beta\text{-}(c,d'))\}.$$

For this, consider the two cases $b \geq c$ and $b < c$ (as in Lemma 6).

Case $b \geq c$: Since $a < q = b + c + 1, \beta(a-1) < \beta(b+c) \leq 2\beta b \leq b$. This, together with P1 implies β-(a,b). Also, $d' < q = b + c + 1$. So, $\frac{\beta}{\beta+1}(d'-1) \leq \frac{\beta}{\beta+1}(b+c-1) = \frac{\beta}{\beta+1}c + \frac{\beta}{\beta+1}(b-1) \leq \frac{\beta}{\beta+1}c + \frac{c}{\beta+1} = c$. This, together with P2 implies $\frac{\beta}{1+\beta}$-(c,d'). So, β-$(a,b) \wedge \frac{\beta}{1+\beta}$-$(c,d')$.

Case $b < c$: Since $a < q = b + c + 1, a - 1 < b + c$. So, $a - 1 \leq b + c - 1$ or $\frac{\beta(a-1)}{1+\beta} \leq \frac{\beta b}{1+\beta} + \frac{\beta(c-1)}{1+\beta} \leq \frac{\beta b}{1+\beta} + \frac{b}{1+\beta} = b$. This and P1 imply $\frac{\beta}{1+\beta}$-(a,b). Also, $d' - 1 \leq q - 2 = b + c - 1$. So, $\beta(d'-1) \leq \beta(b+c-1) < \beta(2c-1) < c$. This and

P2 imply β-(c, d'). Hence, $\frac{\beta}{1+\beta}$-$(a, b) \wedge \beta$-(c, d'). □

The substep(ii) rotations are the same as for insertion.

Theorem 10 *If T is β-balanced, then following a deletion the resulting tree T' is also β-balanced provided* $1/3 \leq \beta \leq \sqrt{2} - 1$.

Proof Similar to that of Theorem 9. □

When $0 < \beta < 1/3$, we need to augment the LL rotation by a transformation for the case $d' = 0$. When $d' = 0, \beta(p-1) \leq d = d' + 1 = 1$. So, $p \leq 1/\beta + 1$ and $gp = p + d' + 1 \leq 1/\beta + 2$. To β-balance at gp, the at most $1/\beta + 2$ nodes in gp are rearranged into any β-BBST in constant time (as $1/\beta + 2$ is a constant). When $d' > 0$, the proof of Lemma 13 part (b) can be changed to show $\beta(c-1) \leq d'$ for $0 < \beta \leq \sqrt{2} - 1$. The new proof is: since $c \leq q, c \leq (p-1)/2$ and $\beta(c-1) \leq \beta(p-1)/2 - \beta \leq d/2 - \beta = d - d/2 - \beta \leq d - 1 - \beta < d'$. The LR rotation needs to be augmented by a transformation for the case $d' = d - 1 < \frac{1}{\beta(2+\beta)} - 1$. At this time, $\beta(p-1) \leq d < \frac{1}{\beta(2+\beta)}$. So, $gp = p + d < \frac{1}{\beta^2(2+\beta)} + 1 + \frac{1}{\beta(2+\beta)}$. To β-balance at gp, we rearrange the fewer than $\frac{1}{\beta^2(2+\beta)} + 1 + \frac{1}{\beta(2+\beta)}$ nodes in the subtree, in constant time, into any β-balanced tree. When $d' \geq \frac{1}{\beta(2+\beta)} - 1$, the proof for $\beta(c-1) \leq d'$ in Lemma 14 needs to be changed to show that the LR substep(i) lemma holds. The new proof is:

$$d \geq \beta(p-1) = \beta(a+b+c+1) \geq \beta(\beta(q-1) + b + c + 1)$$

$$= \beta(\beta(b+c)+b+c+1)$$

$$\geq \beta((1+\beta)\beta(c-1)+(1+\beta)c+1)$$

$$= \beta((1+\beta)^2(c-1)+2+\beta).$$

So, $\beta(c-1) \leq \frac{d-2\beta-\beta^2}{(1+\beta)^2} \leq d-1$ (as $d \geq \frac{1}{\beta(2+\beta)}$) $= d'$.

Also, note that when $\beta = 0$, all trees are β-balanced so the rotations (while not needed) preserve balance.

Theorem 11 *With the special handling of the case $d' = 0$, the tree T' resulting from a deletion in a β-BBST is also β-balanced for $0 \leq \beta \leq \sqrt{2}-1$.*

Lemma 15 *The time needed to delete an element from an n node β-BBST is $O(\log n)$ provided $0 < \beta \leq \sqrt{2}-1$.*

3.4.4 Enhancements

Since our objective is to create search trees with minimum search cost, the rebalancing rotations may be performed at each positioning of *gp* during the backward restructuring pass so long as the conditions for the rotation apply rather than only at *gp* positions where the tree is unbalanced.

Consider Figure 3.4(a). If $p' < d$, then the conditions of Lemmas 5 and 6 cannot apply as $q < p' < d$. However, it is possible that $e > p'$ where e is the size of either the left or right subtree of d. In this case, an RR or RL rotation would reduce the total search cost. The proofs of Lemmas 5 and 6 are easily extended to show that these rotations would preserve balance even though no insertion was done in the subtree

d. The same observation applies to deletion. Hence the backward restructuring pass for the insert and delete operations can determine the need for a rotation at each gp location as below (l and r are, respectively, the left and right children of gp).

if $s(l) > s(r)$ **then** check conditions for an LL and LR rotation

else check conditions for an RR and RL rotation.

The enhanced restructuring procedure used for insertion and deletion is given in Figure 3.10. In the RR and RL cases, we have used the relation '$\geq$' rather than '$>$' as this results in better observed run time.

Since it can be shown that the rotations preserve balance even when there has been no insert or delete, we may check the rotation conditions during a search operation and perform rotations when these improve total search cost.

Finally, we note that it is possible to use other definitions of β-balance. For example, we could require $\beta(s(a) - 2) < s(b)$ and $\beta(s(b) - 2) < s(a)$ for β-(a, b). One can show that the development of this chapter applies to these modifications also. Furthermore, when this new definition is used, the number of comparisons in the second substep of the LR and RL rotations is reduced by one.

3.4.5 Top Down Algorithms

As in the case of red/black and WB(α) trees, it is possible to perform, in $O(\log n)$ time, inserts and deletes using a single top to bottom pass. The algorithms are similar to those already presented.

```
procedure Restructuring ;
begin
while (gp) do
    begin
    if (s(gp.left) > s(gp.right)) then
        begin {check conditions for an LL and LR rotation}
        p = gp.left ;
        if (s(p.left) > s(p.right)) then
            begin if (s(p.left) > s(gp.right)) then do LL rotation; end
        else
            begin
            if (s(p.right) > s(gp.right)) then {LR}
                begin
                do LR rotation ;
                { now notations a, b, c, and d follow from figure 3.1(b) }
                if (β(s(a) − 1) > s(b)) then
                    if ((s(a.right) < (1 + β)s(a.left) + 1 − β) and
                        (s(b) < s(a.left))) then
                        do LL rotation
                    else do LR rotation
                else if (β(s(d) − 1) > s(c)) then
                    if ((s(d.left) < (1 + β)s(d.right) + 1 − β) and
                        (s(c) < s(d.right))) then
                        do RR rotation
                    else do RL rotation ;
                end
            end
        end
    else {check conditions for an RR and RL rotation}
        begin
        p = gp.right ;
        if (s(p.left) > s(p.right)) then
            begin
            if (s(p.left) ≥ s(gp.left)) then {RL}
                do symmetric to the above LR case ;
            end
        else
            begin if (s(p.right) ≥ s(gp.left)) then do RR rotation; end ;
        end ;
    gp = gp.parent ;
    end ;
end ;
```

Figure 3.10. Restructuring procedure

3.5 Simple β-BBSTs

The development of Section 3.4 was motivated by our desire to construct trees with minimal search cost. If instead, we desire only logarithmic performance per operation, we may simplify the restructuring pass so that rotations are performed only at nodes where the β-balance property is violated. In this case, we may dispense with the LL/RR rotations and the first substep of an LR/RL rotation. Only LR/RL substep (ii) rotations are needed. To see this, observe that Lemmas 7 and 11 show that the second substep rotations rebalance at gp (see Figures 3.6 and 3.7) provided $\frac{\beta}{1+\beta}$-(p, d) (The remaining conditions are ensured by the bottom-up nature of restructuring and the fact the tree was β-balanced prior to the insert or delete).

If the operation that resulted in loss of balance at gp was an insert, then $\beta(p-2) \leq d$ (as $p > d$, the insert took place in subtree p and gp was β-balanced prior to the insert) and $\beta(p-1) > d$ (gp is not β-balanced following the insert). For the substep (ii) rotation to restore balance, we need $\beta(p-1) \leq (1+\beta)d$. This is assured if $d + \beta \leq (\beta+1)d$ (as $\beta(p-2) \leq d$). So, we need $d \geq 1$. If $d < 1$, then $d = 0$. Now $\beta(p-2) \leq d$ and $\beta(p-1) > d$ imply $p = 2$. One may verify that when $p = 2$, the LR(ii) rotations restore balance.

If the loss of β-balance at gp is the result of a deletion (say from its right subtree), then $\beta(p-1) \leq d+1$ (as gp was β-balanced prior to the delete). For the substep (ii) rotation to accomplish the rebalancing, we need $\beta(p-1) \leq (\beta+1)d$. This is guaranteed if $d+1 \leq (\beta+1)d$ or $d \geq 1/\beta$. When $d < 1/\beta$ and $\beta \geq 1/3$, $d \leq 2$. Since $\beta(p-1) \leq d+1$ and $\beta \geq 1/3$, when $d = 2$, $p \leq 10$; when $d = 1$, $p \leq 7$; and when

$d = 0$, $p \leq 4$. We may verify that for all these cases, the LR(ii) rotations restore balance. Hence, the only problematic case is when $\beta < 1/3$ and $d < 1/\beta$.

When $\beta < 1/3$, an LL rotation fails to restore balance only when $d = 0$ (see discussion following Theorem 10). So we need to rearrange the at most $1/\beta + 2$ nodes in gp into any β-balanced tree when $d = 0$. An LR rotation fails only when $d < \frac{1}{\beta(2+\beta)} - 1$. To see this, note that in the terminology of Lemma 14, d is d'. The proof of P2 is extended to the case $\beta \leq 1/3$ when $d' \geq \frac{1}{\beta(2+\beta)} - 1$. Also, since $d' < 1/\beta$, for the case $b \geq c$, we get $\beta(d' - 1) < 1 - \beta < c$ (as $c \geq 1$). For the case $b < c$, we need to show $\beta(a - 1) \leq b$. Since an LR rotation is done only when condition $D1 \vee D2$ holds, from Lemmas 10 and 11, it follows that $\beta(a - 1) \leq b$. So, an LR rotation rebalances when $\beta < 1/3$ provided $d \geq \frac{1}{\beta(2+\beta)} - 1$. For smaller d, the at most $\frac{1}{\beta^2(2+\beta)} + \frac{1}{\beta(2+\beta)} + 1$ nodes in the subtree gp may be directly rearranged into a β-balanced tree.

The restructuring algorithm for simple β-BBSTs is given in Figures 3.11 and 3.12. The algorithm of Figure 3.11 is used following an insert and that of Figure 3.12 after a delete.

Simple β-BBSTs are expected to have higher search cost than the β-BBSTs of Section 3.4. However, they are a good alternative to traditional WB(α) trees as they are expected to be "better balanced". To see this, note that from the proof of Lemma 3, the balance, $B(p)$, at any node p in a β-balanced tree satisfies

$$\frac{1}{B(p)} = 1 + \frac{s(r) + 1}{s(l) + 1}$$

```
procedure Restructuring2 ;
begin
while (gp) do
    begin
    if (β(s(gp.left) − 1) > s(gp.right)) then {do an LL or LR rotation}
        begin
        p = gp.left ;
        if ((s(p.right) < (1 + β)s(p.left) + 1 − β) and
            (s(gp.right) < s(p.left))) then
            do LL rotation
        else do LR rotation ;
        end
    else
        do symmetric to the above L case ;
    gp = gp.parent ;
    end ;
end ;
```

Figure 3.11. Simple restructuring procedure for insertion

```
procedure Restructuring3 ;
begin
while (gp) do
    begin
    if (β(s(gp.left) − 1) > s(gp.right)) then
        if (β < 1/3) and (s(gp.right) < 1/β(2 + β) − 1) then
            rearrange the subtree rooted at gp into any β-balanced tree
        else {do an LL or LR rotation}
            begin
            p = gp.left ;
            if ((s(p.right) < (1 + β)s(p.left) + 1 − β) and
                (s(gp.right) < s(p.left))) then
                do LL rotation
            else do LR rotation ;
            end
        end
    else
        do symmetric to the above L case ;
    gp = gp.parent ;
    end ;
end ;
```

Figure 3.12. Simple restructuring procedure for deletion

$$\begin{aligned} &\geq 1 + \frac{1}{1/\beta + \frac{2\beta-1}{\beta(s(r)+1)}} \\ &= \frac{1 + \frac{1}{\beta} + \frac{2\beta-1}{\beta(s(r)+1)}}{\frac{1}{\beta} + \frac{2\beta-1}{\beta(s(r)+1)}}. \end{aligned}$$

So,

$$B(p) \leq 1 - \frac{1}{1 + \frac{1}{\beta} + \frac{2\beta-1}{\beta(s(r)+1)}}.$$

Also, since $s(r) - 1 \leq s(l)/\beta$, $s(r) + 1 \leq s(l)/\beta + 2$. Hence, $1 + \frac{s(r)+1}{s(l)+1} \leq 1 + \frac{s(l)}{\beta(s(l)+1)} + \frac{2}{s(l)+1}$. So,

$$\begin{aligned} B(p) &\geq \frac{1}{1 + \frac{1}{\beta} - \frac{1}{\beta(s(l)+1)} + \frac{2}{s(l)+1}} \\ &= \frac{1}{1 + \frac{1}{\beta} + \frac{2\beta-1}{\beta(s(l)+1)}}. \end{aligned}$$

Consequently,

$$\frac{1}{1 + \frac{1}{\beta} + \frac{2\beta-1}{\beta(s(l)+1)}} \leq B(p) \leq 1 - \frac{1}{1 + \frac{1}{\beta} + \frac{2\beta-1}{\beta(s(r)+1)}}.$$

When $\beta = \sqrt{2} - 1$,

$$\frac{1}{2 + \sqrt{2} + \frac{1-\sqrt{2}}{s(l)+1}} \leq B(p) \leq 1 - \frac{1}{2 + \sqrt{2} + \frac{1-\sqrt{2}}{s(r)+1}}.$$

If $s(p) \leq 10$, $0.296 \leq B(p) \leq 1 - 0.296$. So, every β-balanced subtree with 10 or fewer nodes is in WB(α) for $\alpha \approx 0.296$. Similarly, every subtree with 100 or fewer nodes is in WB(α) for $\alpha \approx 0.293$. In fact, for every fixed k, subtrees of size k or less

```
procedure Restructuring4 ;
begin
while (gp) do
    begin
    if (s(gp.left) > s(gp.right)) then
        begin  {check conditions for an LL and LR rotation}
        p = gp.left ;
        if (s(p.left) > s(p.right)) and (s(p.left) > s(gp.right)) then
            do LL rotation
        else if (s(p.left) ≤ s(p.right)) and (s(p.right) > s(gp.right)) then
            do LR rotation ;
        end
    else {check conditions for an RR and RL rotation}
        do symmetric to the above L case ;
    gp = gp.parent ;
    end ;
end ;
```

Figure 3.13. Simple restructuring procedure without a β value

are in WB(α) for α slightly higher than $1 - \frac{1}{\sqrt{2}} \approx 0.2929$ which is the largest value of α for which WB(α) trees can be maintained.

3.6 BBSTs without Deletion

In some applications of a dictionary, we need to support only the insert and search operations. In these applications, we can construct binary search trees with total cost

$$C(T) \leq n \log_{\phi}(\sqrt{5}(n+1))$$

by using the simpler restructuring algorithm of Figure 3.13.

Theorem 12 *When the only operations are search and insert and restructuring is done as in Figure 3.13,* $C(T) \leq n \log_{\phi}(\sqrt{5}(n+1))$.

Proof Suppose T currently has $m-1$ elements and a new element is inserted. Let u be the level at which the new element is inserted. Suppose that the restructuring pass performs rotations at $q < u$ of the nodes on the path from the root to the newly inserted node. Then $C(T)$ increases by at most $v = u - q$ as a result of the insertion. The number of nodes on the path from the root to the newly inserted node at which no rotation is performed is also v. Let these nodes be numbered 1 through v bottom to top. Let S_i denote the number of elements in the subtree with root i prior to the restructuring pass. We see that $S_1 \geq 1$ and $S_2 \geq 2$. For node i, $2 < i \leq v$, one of its subtrees contains node $i-1$. Without loss of generality, let this be the left subtree of i. Let the root of the right subtree of i be d. So,

$$S_i \geq S_{i-1} + s(d) + 1.$$

If $i-1$ is not the left child of i, then since no rotation is done at i, $s(d) \geq S_{i-1}$. If $i-1$ is the left child of i, then consider node $i-2$. This is in one of the subtrees of i. Since no rotation is performed at $i-1$, $s(d) \geq S_{i-2}$. Since $S_{i-1} > S_{i-2}$, we get

$$S_i \geq S_{i-1} + S_{i-2} + 1.$$

Hence, $S_v \geq N_v$ where N_v is the minimum number of elements in a COST of height v. So, $v \leq \log_\phi(\sqrt{5}(m+1))$. So, when an element is inserted into a tree that has $m-1$ elements, its cost $C(T)$ increases by at most $\log_\phi(\sqrt{5}(m+1))$. Starting with an empty tree and inserting n elements results in a tree whose cost is at most

$n \log_\phi(\sqrt{5}(n+1))$. □

Corollary 2 *The expected cost of a search or insert in a BBST constructed as above is $O(\log n)$.*

Proof Since $C(T) \leq n \log_\phi(\sqrt{5}(n+1))$, the expected search cost is $C(T)/n \leq \log_\phi(\sqrt{5}(n+1))$. The cost of an insert is the same order as that of a search as each insert follows the corresponding search path twice (top down and bottom up). □

3.7 Experimental Results

For comparison purposes, we wrote C programs for BBSTs, SBBSTs (simple BBSTs), BBSTDs (BBSTs in which procedure Restructuring4 (Figure 3.13) is used to restructure following inserts as well as deletes), unbalanced binary search trees (BST), AVL-trees, top-down red-black trees (RB-T), bottom-up red-black trees (RB-B) [31], weight balanced trees (WB), deterministic skip lists (DSL), treaps (TRP), and skip lists (SKIP). For the BBST and SBBST structures, we used $\beta = 207/500$ while for the WB structure, we used $\alpha = 207/707$. While these are not the highest permissible values of β and α, this choice permitted us to use integer arithmetic rather than the substantially more expensive real arithmetic. For instance, β-(a, b) for $\beta = 207/500$ can be checked using the comparisons $207(s(a) - 1) > 500s(b)$ and $207(s(b) - 1) > 500s(a)$. The randomized structures TRP and SKIP used the same random number generator with the same seed. SKIP was programmed with probability value $p = 1/4$ as in Pugh [26].

To minimize the impact of system call overheads on run time measurements, we programmed all structures using simulated pointers (i.e., an array of nodes with integer pointers [27]. Skip lists use variable size nodes. This requires more complex storage management than required by the remaining structures which use nodes of the same size. For our experiments, we implemented skip lists using fixed size nodes, each node being of the maximum size. As a result, our run times for skip lists are smaller than if a space efficient implementation had been used. In all our tree structure implementations, null pointers were replaced by a pointer to a tail node whose data field could be set to the search/insert/delete key and thus avoid checking for falling off the tree. Similar tail pointers are part of the defined structure of skip and deterministic skip lists. Each tree also had a head node. WB(α) trees were implemented with a bottom-up restructuring pass. Our codes for SKIP and DSL are based on the codes of Pugh [26] and Papadakis [22], respectively. Our AVL and RB-T codes are based on those of Papadakis [22] and Sedgewick [28]. The treap structure was implemented using joins and splits rather than rotations. This results in better performance. Furthermore, AVL, RB-B, WB, and BBST were implemented with parent pointers in addition to left and right child pointers. For BBSTs, the enhancements described in Section 3.4.4 for insert and delete (see Figure 3.10) were employed. No rotations were performed during a search when using any of the structures.

For our experiments, we tried two versions of the code. These varied in the order in which the 'equality' and 'less than' or 'greater than' check between x and e (where x is the key being searched/inserted/deleted and e is the key in the current

node) is done. In version 1, we conducted an initial experiment to determine if the total comparison count is less using the order L:

if $x < e$ **then** move to left child

else if $x \neq e$ **then** move to right child

else found

or the order R:

if $x > e$ **then** move to right child

else if $x \neq e$ **then** move to left child

else found.

Our experiment indicated that doing the 'left child' check first (i.e. order L) worked better for AVL, BBST, BBSTD, and DSL structures while R worked better for the RB-T, RB-B, WB, SBBST, and TRP structures. No significant difference between L and R was observed for BSTs. For skip lists, we do not have the flexibility to change the comparison order. The version 1 codes performed the comparisons in the order determined to be better. For BSTs, the order R was used.

In the version 2 codes the comparisons in each node took the standard form

if $x = e$ **then** found

else if $x < e$ **then** move to left child

else move to right child.

The version 2 restructuring code for BBSTs differed from that of Figure 3.10 in that the '$>$' test in the second, third, and forth **if** statements was changed to '$\geq$'.

No change was made in the corresponding if statements for RR and RL rotations. While this increased the number of comparisons, it reduced the run time.

We experimented with n = 10,000, 50,000, 100,000, and 200,000. For each n, the following experiments were conducted:

(a) start with an empty structure and perform n inserts;

(b) search for each item in the resulting structure once; items are searched for in the order they were inserted

(c) perform an alternating sequence of n inserts and n deletes; in this, the n elements inserted in (a) are deleted in the order they were inserted and n new elements are inserted

(d) search for each of the remaining n elements in the order they were inserted

(e) delete the n elements in the order they were inserted.

For each n, the above five part experiment was repeated ten times using different random permutations of distinct elements. For each permutation, we measured the total number of element comparisons performed and then averaged these over the ten permutations.

First, we report on the relative performance of SBBSTs, BBSTDs, and BBSTs. For this comparison, we used only version 1 of the code. Table 3.1 gives the average number of key comparisons performed for each of the five parts of the experiment. The three versions of our proposed data structure are very competitive on this measure. BBSTDs and BBSTs generally performed fewer comparisons than did SBBSTs. All three structures had a comparison count within 2% of one another.

Table 3.1. The number of key comparisons on random inputs (version 1 code)

n	operation	SBBST	BBSTD	BBST
10,000	insert	212495	212223	212111
	search	194661	191599	191578
	ins/del	416125	416967	416862
	search	194957	191666	191676
	delete	168033	166441	166487
50,000	insert	1241080	1236682	1236114
	search	1152137	1135131	1134969
	ins/del	2437918	2438083	2437639
	search	1153821	1134277	1134062
	delete	1018675	1007766	1007688
100,000	insert	2635913	2624829	2623792
	search	2458079	2423988	2423613
	ins/del	5183619	5180383	5179653
	search	2461221	2420282	2419990
	delete	2190798	2168049	2168110
200,000	insert	5580139	5555190	5553256
	search	5223989	5148220	5147698
	ins/del	10981441	10969578	10968053
	search	5229172	5144808	5144148
	delete	4692447	4641349	4641389

However, when we used ordered data rather than random data (Table 3.2), SBBSTs performed noticeably inferior to BBSTDs and BBSTs; the later two remained very competitive.

Tables 3.3 and 3.4 give the average heights of the trees using random data and using ordered data, respectively. The first number gives the height following part (a) of the experiment and the second following part (c). The numbers are identical for BBSTDs and BBSTs and slightly higher (lower) for SBBSTs using random (ordered) data.

Table 3.2. The number of key comparisons on ordered inputs (version 1 code)

n	operation	SBBST	BBSTD	BBST
10,000	insert	170182	150554	150554
	search	188722	185530	185530
	ins/del	425305	315177	314998
	search	191681	184155	184155
	delete	215214	135311	135131
50,000	insert	991526	872967	872967
	search	1117174	1101481	1101481
	ins/del	2472808	1806346	1805439
	search	1116390	1098065	1098065
	delete	1277756	792717	791815
100,000	insert	2103808	1850548	1850548
	search	2384327	2354757	2354757
	ins/del	5249194	3823415	3821594
	search	2382759	2346118	2346128
	delete	2738294	1686397	1684584
200,000	insert	4449143	3903083	3903083
	search	5068632	4946753	4946753
	ins/del	11105525	8051695	8048058
	search	5065496	5001967	5001967
	delete	5842168	3580856	3577223

Table 3.3. Height of the trees on random inputs (version 1 code)

n	SBBST	BBSTD	BBST
10,000	17,17	16,16	16,16
50,000	20,20	19,19	19,19
100,000	21,21	20,20	20,20
200,000	22,23	21,21	21,21

Table 3.4. Height of the trees on ordered inputs (version 1 code)

n	SBBST	BBSTD	BBST
10,000	16,15	17,17	17,17
50,000	20,20	20,20	20,20
100,000	21,21	21,21	21,21
200,000	22,22	23,22	23,22

The average number of rotations performed by each of the three structures is given in Tables 3.5 and 3.6. A single rotation (i.e., LL or RR) is denoted 'S' and a double rotation (i.e., LR or RL) denoted 'D'. In the case of BBSTs, double rotations have been divided into three categories: D = LR and RL rotations that do not perform a second substep rotation; DS = LR and RL rotations with a second substep rotation of type LL and RR; DD = LR and RL rotations with a second substep rotation of type LR and RL. BBSTDs and BBSTs performed a comparable number of rotations on both data sets. However, on random data SBBSTs performed about half as many rotations as did BBSTDs and BBSTs. On ordered data, SBBSTs performed 15 to 20% fewer rotations on part (a), 34% fewer on part (c), and 51% fewer on part (e).

The run-time performance of the structures is significantly influenced by compiler and architectural features as well as the complexity of a key comparison. The results we report are from a SUN SPARC-5 using the UNIX C compiler cc with optimization option. Because of instruction pipelining features, cache replacement policies, etc., the measured run times are not always consistent with the compiler and architecture independent metrics reported in Tables 3.1 through 3.6 and later in Tables 3.11 through 3.16. For example, since the search codes for all tree based methods are essentially identical, we would expect methods with a smaller comparison count to have a smaller run time for parts (b) and (d) of the experiment. This was not always the case.

Tables 3.7 and 3.8 give the run times of the three BBST structures using integer keys and Tables 3.9 and 3.10 do this for the case of real (i.e., floating point) keys. The

Table 3.5: The number of rotations on random inputs (version 1 code)

		SBBST		BBSTD		BBST			
n	operation	S	D	S	D	S	D	DS	DD
10,000	insert	2341	2220	5045	4314	5025	3938	151	93
	ins/del	4269	3216	10158	6311	10104	5849	232	103
	delete	1607	1110	5235	2104	5201	2018	51	28
50,000	insert	11719	11120	25216	21596	25059	19732	754	455
	ins/del	21330	16125	51238	31499	50979	29198	1161	531
	delete	8058	5648	26214	10462	26068	10033	248	131
100,000	insert	23450	22262	50283	43230	50047	39461	1527	920
	ins/del	42780	32203	102218	62967	101836	58491	2275	1046
	delete	16095	11306	52227	21022	51943	20147	496	260
200,000	insert	46934	44525	100664	86605	100205	79013	3054	1840
	ins/del	85283	64417	204459	125960	203568	116940	4593	2059
	delete	32233	22551	104344	41884	103826	40157	990	523

Table 3.6. The number of rotations on ordered inputs (version 1 code)

		SBBST		BBSTD		BBST			
n	operation	S	D	S	D	S	D	DS	DD
10,000	insert	9984	0	9985	2387	9985	2387	0	0
	ins/del	14997	0	16567	6130	16644	5797	25	154
	delete	4989	0	6570	3726	6647	3392	26	154
50,000	insert	49980	0	49983	11956	49983	11956	0	0
	ins/del	74996	0	82862	30659	83247	28982	137	770
	delete	24987	0	32859	18686	33242	17018	136	766
100,000	insert	99979	0	99983	23917	99983	23917	0	0
	ins/del	149996	0	165738	61327	166504	57969	280	1540
	delete	49986	0	65733	37392	66505	34040	278	1536
200,000	insert	199978	0	199982	47839	199982	47839	0	0
	ins/del	299996	0	331473	122653	333012	115938	559	3078
	delete	99985	0	131478	74795	133016	68086	557	3076

sum of the run time for parts (a) – (e) of the experiment is graphed in Figure 3.14. For random data, SBBSTs significantly and consistently outperformed BBSTDs and BBSTs. On ordered data, however, BBSTDs were slightly faster than BBSTs and both were significantly faster than SBBSTs.

Since BBSTs generated trees with the least search cost, we expect BBSTs to outperform SBBSTs and BBSTDs in applications where the comparison cost is very high relative to that of other operations and searches are done with a much higher frequency than inserts and deletes. However, with the mix of operations used in our tests, SBBSTs are the clear choice for random inputs and BBSTDs for ordered inputs.

In comparing with the other structures, our tables repeat the data for BBSTs. The reader may make the comparison with SBBSTs and BBSTDs.

Table 3.7. Run time on random inputs using integer keys (version 1 code)

n	operation	SBBST	BBSTD	BBST
10,000	insert	0.27	0.30	0.34
	search	0.06	0.06	0.07
	ins/del	0.57	0.62	0.70
	search	0.06	0.06	0.06
	delete	0.22	0.25	0.26
50,000	insert	1.48	1.61	1.75
	search	0.35	0.36	0.37
	ins/del	2.90	3.47	3.84
	search	0.36	0.38	0.39
	delete	1.13	1.47	1.62
100,000	insert	3.00	3.57	3.80
	search	0.78	0.83	0.84
	ins/del	6.28	7.78	8.41
	search	0.83	0.87	0.88
	delete	2.54	3.31	3.58
200,000	insert	6.56	7.74	8.37
	search	1.80	1.89	1.89
	ins/del	13.89	17.32	18.57
	search	1.86	1.98	1.98
	delete	5.64	7.41	8.02

Time Unit : *sec*

Table 3.8. Run time on ordered inputs using integer keys (version 1 code)

n	operation	SBBST	BBSTD	BBST
10,000	insert	0.32	0.20	0.27
	search	0.05	0.03	0.05
	ins/del	0.58	0.43	0.57
	search	0.07	0.03	0.03
	delete	0.20	0.17	0.23
50,000	insert	1.38	1.20	1.10
	search	0.25	0.20	0.20
	ins/del	2.63	2.18	2.40
	search	0.25	0.20	0.20
	delete	0.95	0.92	1.05
100,000	insert	3.43	2.23	2.53
	search	0.72	0.45	0.42
	ins/del	5.97	4.70	5.13
	search	0.55	0.47	0.42
	delete	2.10	1.98	2.15
200,000	insert	6.65	4.95	5.25
	search	1.20	0.92	0.90
	ins/del	13.13	10.23	10.88
	search	1.17	0.90	0.90
	delete	4.63	4.25	4.58

Time Unit : *sec*

Table 3.9. Run time on random real inputs (version 1 code)

n	operation	SBBST	BBSTD	BBST
10,000	insert	0.23	0.34	0.36
	search	0.07	0.10	0.10
	ins/del	0.44	0.75	0.79
	search	0.08	0.10	0.10
	delete	0.17	0.29	0.30
50,000	insert	1.43	1.76	1.93
	search	0.47	0.53	0.52
	ins/del	2.76	3.89	4.22
	search	0.50	0.54	0.55
	delete	1.13	1.62	1.76
100,000	insert	2.96	3.94	4.36
	search	1.08	1.17	1.16
	ins/del	6.11	8.58	9.30
	search	1.12	1.20	1.22
	delete	2.50	3.66	3.95
200,000	insert	6.85	8.92	9.33
	search	2.41	2.58	2.57
	ins/del	13.86	19.49	20.46
	search	2.49	2.69	2.66
	delete	5.61	8.25	8.80

Time Unit : *sec*

Table 3.10. Run time on ordered real inputs (version 1 code)

n	operation	SBBST	BBSTD	BBST
10,000	insert	0.27	0.23	0.20
	search	0.08	0.07	0.07
	ins/del	0.53	0.50	0.43
	search	0.08	0.07	0.05
	delete	0.18	0.23	0.20
50,000	insert	1.43	1.25	1.12
	search	0.40	0.30	0.30
	ins/del	2.80	2.17	2.37
	search	0.40	0.30	0.30
	delete	1.07	0.90	0.97
100,000	insert	3.28	2.58	2.77
	search	0.90	0.62	0.63
	ins/del	6.15	4.70	5.13
	search	0.87	0.62	0.63
	delete	2.35	1.93	2.10
200,000	insert	7.37	4.55	4.92
	search	1.85	1.32	1.32
	ins/del	13.35	10.03	10.93
	search	1.87	1.33	1.33
	delete	5.08	4.17	4.43

Time Unit : *sec*

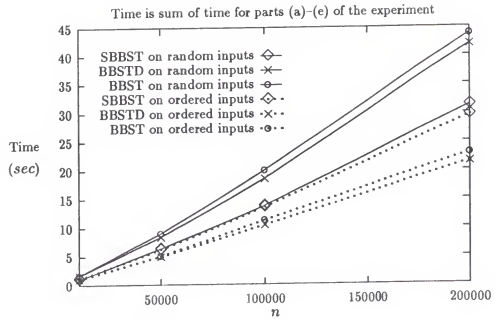


Figure 3.14. Run time on real inputs (version 1 code)

The average number of comparisons for each of the five parts of the experiment are given in Table 3.11 for the version 1 implementation. On the comparison measure, AVL, RB-B, WB, and BBSTs are the front runners and are quite competitive with one another. On parts (a) (insert n elements) and (c) (insert n and delete n elements), AVL trees performed best while on the two search tests ((b) and (d)) and the deletion test (e), BBSTs performed best.

Table 3.12 gives the number of comparisons performed when ordered data (i.e., the elements in part (a) are $1, 2, \ldots, n$ and are inserted in this order) and those in part (c) are $n+1, \ldots, 2n$ (in this order) is used instead of random permutations of distinct elements. This experiment attempts to model realistic situations in which the inserted elements are in "nearly sorted order". BSTs were not included in this test as they perform very poorly with ordered data taking $O(n^2)$ time to insert n

times. The computer time needed to perform this test on BSTs was determined to be excessive. This test exhibited greater variance in performance. Among the deterministic structures, BBSTs outperformed the others in parts (a) – (d) while AVL trees were ahead in part (e). For part (a), BBSTs performed approximately 45% fewer comparisons than did AVL trees and approximately 12% fewer than WB trees. The randomized structure TRP was the best of the eight structures reported in Table 3.12 for part (a). It performed approximately 10% fewer comparisons than did BBST trees. However, the BBST remained best overall on parts (b), (c), and (d).

The heights of the trees (number of levels in the case of DSL and SKIP) for the experiments with random and ordered data are given in Tables 3.13 and 3.14 respectively. The first number in each table entry is the tree height after part (a) of the experiment and the second, the height after part (c). In all cases, the number of levels using skip lists is fewest. However, among the tree structures, AVL and BBST trees have least height on random data and AVL has least with ordered data.

Tables 3.15 and 3.16, respectively, give the number of rotations performed by each of the deterministic tree schemes for experiment parts (a), (c), and (e). Note that none of the schemes performs rotations during a search.

On ordered data, BBSTs perform about 25% more rotations than do the remaining structures. These remaining structures perform about the same number of rotations. On random data, AVL trees, bottom-up red-black trees and WB trees perform a comparable number of rotations. Top-down red-black trees and BBST trees

Table 3.11: The number of key comparisons on random inputs (version 1 code)

n	operation	BST	AVL	RB-T	RB-B	WB	BBST	DSL	TRP	SKIP
	insert	264175	211401	262838	211886	211916	212111	276247	296866	224757
	search	254175	193253	194606	194291	194153	191578	258089	258662	255072
10,000	ins/del	516853	411220	515184	414990	414635	416862	923524	601137	519430
	search	252200	193141	197399	195525	194442	191676	256578	254119	256124
	delete	215555	167312	200218	167455	167531	166487	526242	242743	231745
	insert	1560958	1234911	1550701	1236968	1238628	1236114	1640660	1717037	1357076
	search	1510958	1147273	1150466	1146754	1149970	1134969	1512093	1503452	1537547
50,000	ins/del	3061868	2417733	3058045	2424944	2431281	2437639	5351715	3456045	2996512
	search	1500504	1145808	1173662	1152764	1151578	1134062	1499657	1497081	1501731
	delete	1316917	1013535	1242426	1013144	1015988	1007688	3077266	1451835	1373858
	insert	3329780	2623894	3305332	2626314	2631411	2623792	3513401	3632046	2919371
	search	3229780	2445659	2451137	2446466	2453855	2423613	3244497	3247143	3188621
100,000	ins/del	6537563	5137280	6564352	5154118	5170695	5179653	11545200	7476441	6399463
	search	3208453	2443038	2502098	2457531	2456748	2419990	3229747	3310823	3225343
	delete	2839934	2181327	2692672	2177946	2185213	2168110	6561272	3177135	2981173
	insert	7076132	5553640	7016676	5558174	5571133	5553256	7483199	7682439	6178596
	search	6876132	5191730	5209189	5199786	5215568	5147698	6887196	6797942	6697223
200,000	ins/del	13907058	10862426	13940982	10921880	10956496	10968053	24207106	15543559	13377747
	search	6830718	5186737	5332771	5223154	5220965	5144148	6814733	6916150	6680642
	delete	6095324	4664876	5800203	4664344	4680768	4641389	13811271	6700557	6149268

Table 3.12: The number of key comparisons on ordered inputs (version 1 code)

n	operation	AVL	RB-T	RB-B	WB	BBST	DSL	TRP	SKIP
10,000	insert	277234	376228	241383	171017	150554	435199	135989	247129
	search	191917	188246	190106	188722	185530	262423	271087	256706
	ins/del	421032	718040	508810	425843	314998	983676	390899	354566
	search	195133	189494	190090	191681	184155	249694	269031	250538
	delete	104038	276136	218216	214930	135131	468244	193080	84392
50,000	insert	1618930	2233658	1436225	995720	872967	2585557	825390	1422120
	search	1120497	1117001	1120495	1117174	1101481	1509152	1540082	1467217
	ins/del	2418422	4311748	3055100	2475487	1805439	6019215	2194668	1973416
	search	1124001	1168633	1126126	1116390	1098065	1481819	1568903	1449810
	delete	607478	1719212	1323918	1276262	791815	2785792	1181612	486498
100,000	insert	3437858	4767564	3072389	2112201	1850548	5521408	1724473	2925618
	search	2390963	2383979	2390961	2384327	2354757	3218246	3564282	2970715
	ins/del	5111850	9223606	6510188	5254541	3821594	12788447	4438266	4406427
	search	2397971	2487243	2402224	2382759	2346128	3163554	3281308	3277089
	delete	1289954	3737982	2847792	2735270	1684584	5971196	2403622	961283
200,000	insert	7275714	10135418	6544713	4465935	3903083	11743159	3428355	6403207
	search	5081893	5067933	5081891	5068632	4946753	6836428	7174727	6448304
	ins/del	10773706	19647336	13820364	11116226	8048058	27076911	9054078	9062233
	search	5095909	5274461	5104418	5065496	5001967	6727017	7006341	6458321
	delete	2729906	8075474	6095538	5836096	3577223	12741948	5094044	1995215

Table 3.13. Height of the trees on random inputs (version 1 code)

n	BST	AVL	RB-T	RB-B	WB	BBST	DSL	TRP	SKIP
10,000	31,31	16,16	17,18	16,17	17,17	16,16	12,11	32,31	8,8
50,000	38,38	19,19	20,21	19,20	20,20	19,19	13,12	38,37	9,9
100,000	41,41	20,20	21,22	20,21	21,22	20,20	14,13	41,40	9,9
200,000	44,43	21,21	22,24	21,22	23,23	21,21	15,14	43,44	9,9

Table 3.14. Height of the trees on ordered inputs (version 1 code)

n	AVL	RB-T	RB-B	WB	BBST	DSL	TRP	SKIP
10,000	14,14	20,20	24,24	16,15	17,17	14,13	33,34	8,8
50,000	16,16	23,23	29,28	20,20	20,20	16,16	41,41	9,9
100,000	17,17	25,25	31,30	21,21	21,21	17,17	46,41	9,9
200,000	18,18	27,27	33,32	22,22	23,22	18,18	47,46	9,9

perform a significantly larger number of rotations. In fact, BBSTs perform about twice as many rotations as AVL trees.

The average run times for the random data tests are given in Table 3.17 and in Table 3.18 for the ordered data test. Both of these use integer keys. The times using real keys are given in Tables 3.19 and 3.20. The sum of the run time for parts (b) and (d) of the experiment is graphed in Figure 3.15 for random data and in Figure 3.16 for ordered data. The graph of Figure 3.17 shows only one line MIX for AVL, RB-T, RB-B, WB, and BBST while that of Figure 3.18 shows MIX for AVL, RB-T, RB-B, and WB as the times for these are very close. With integer keys and random data, unbalanced binary search trees (BSTs) outperformed each of the remaining structures. The next best performance was exhibited by bottom-up red-black trees. They did marginally better than AVL trees. The remaining

Table 3.15: The number of rotations on random inputs (version 1 code)

n	operation	AVL		RB-T		RB-B		WB		BBST			
		S	D	S	D	S	D	S	D	S	D	DS	DD
10,000	insert	2328	2322	1964	1955	1946	1933	2274	2065	5025	3938	151	93
	ins/del	4343	3224	14773	8213	4053	2591	4256	2978	10104	5849	232	103
	delete	1645	1120	9558	2678	1845	1166	1595	1022	5201	2018	51	28
50,000	insert	11664	11614	9822	9815	9710	9689	11355	10352	25059	19732	754	455
	ins/del	21585	16214	81895	45180	20255	12979	21266	14975	50979	29198	1161	531
	delete	8231	5630	54806	13431	9196	5844	7963	5194	26068	10033	248	131
100,000	insert	23316	23254	19593	19677	19340	19414	22723	20730	50047	39461	1527	920
	ins/del	43243	32361	196769	103835	40618	25919	42567	29898	101836	58491	2275	1046
	delete	16466	11264	119825	26953	18530	11708	16024	10420	51943	20147	496	260
200,000	insert	46631	46518	39290	39291	38797	38793	45458	41480	100205	79013	3054	1840
	ins/del	86218	64712	394187	209941	80892	52030	84927	59911	203568	116940	4593	2059
	delete	33047	22477	247905	54046	37083	23379	31984	20800	103826	40157	990	523

Table 3.16: The number of rotations on ordered inputs (version 1 code)

		AVL		RB-T		RB-B		WB		BBST			
n	operation	S	D	S	D	S	D	S	D	S	D	DS	DD
10,000	insert	9986	0	9980	0	9976	0	9984	0	9985	2387	0	0
	ins/del	14996	0	14999	0	14995	0	14997	0	16644	5797	25	154
	delete	4990	0	4983	1	4989	0	4989	0	6647	3392	26	154
50,000	insert	49984	0	49977	0	49971	0	49980	0	49983	11956	0	0
	ins/del	74994	0	75000	0	74994	0	74996	0	83247	28982	137	770
	delete	24988	0	24978	1	24986	0	24987	0	33242	17018	136	766
100,000	insert	99983	0	99975	0	99969	0	99979	0	99983	23917	0	0
	ins/del	149994	0	150000	0	149994	0	149996	0	166504	57969	280	1540
	delete	49987	0	49977	1	49985	0	49986	0	66505	34040	278	1536
200,000	insert	199982	0	199973	0	199967	0	199978	0	199982	47839	0	0
	ins/del	299994	0	300000	0	299994	0	299996	0	333012	115938	559	3078
	delete	99986	0	99976	1	99984	0	99985	0	133016	68086	557	3076

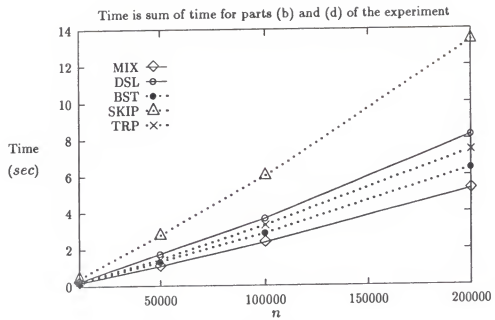


Figure 3.15. Run time on random real inputs (version 1 code)

structures have a noticeably inferior structure. For ordered integer keys, BSTs take more time than we were willing to expend. Of the remaining structures, treaps generally performed best on parts (a), (c), and (e) while BBSTs did best on parts (b) and (d).

With real keys and random data, BSTs did not outperform the remaining structures. Now, the five balanced binary tree structure became quite competitive with respect to the search operations (i.e., parts (b) and (d)). RB-B generally outperformed the other structures on parts (a), (c), and (e). Using ordered real keys, the treap was the clear winner on parts (a), (c), and (e) while BBSTs handily outperformed the remaining structures on parts (b) and (d).

Some of the experimental results using version 2 of the code are shown in Tables 3.21– 3.24. On the comparison measure, with random data (Table 3.21), skip

Table 3.17: Run time on random inputs using integer keys (version 1 code)

n	operation	BST	AVL	RB-T	RB-B	WB	BBST	DSL	TRP	SKIP
10,000	insert	0.08	0.12	0.15	0.12	0.20	0.34	0.19	0.18	0.24
	search	0.05	0.05	0.05	0.06	0.05	0.07	0.09	0.09	0.18
	ins/del	0.14	0.21	0.36	0.22	0.39	0.70	0.49	0.33	0.45
	search	0.05	0.05	0.05	0.05	0.05	0.06	0.09	0.09	0.18
	delete	0.05	0.08	0.12	0.09	0.16	0.26	0.20	0.08	0.16
50,000	insert	0.65	0.79	0.98	0.73	1.18	1.75	1.10	1.01	1.36
	search	0.40	0.36	0.36	0.36	0.35	0.37	0.58	0.56	1.25
	ins/del	1.04	1.48	2.50	1.26	2.22	3.84	2.77	1.86	2.73
	search	0.40	0.41	0.44	0.36	0.36	0.39	0.57	0.56	1.16
	delete	0.39	0.54	1.01	0.51	0.94	1.62	1.16	0.51	1.10
100,000	insert	1.34	1.57	2.10	1.54	2.54	3.80	2.46	2.23	2.84
	search	0.88	0.80	0.80	0.83	0.78	0.84	1.36	1.30	2.63
	ins/del	2.36	3.21	5.52	2.74	4.86	8.41	6.35	4.10	6.13
	search	0.93	0.94	1.00	0.84	0.83	0.88	1.33	1.29	2.61
	delete	0.88	1.24	2.26	1.14	2.11	3.58	2.64	1.23	2.41
200,000	insert	2.79	3.37	4.41	3.18	5.21	8.37	5.56	4.70	6.25
	search	2.00	1.80	1.81	1.81	1.78	1.89	3.03	2.91	5.85
	ins/del	5.24	6.99	12.51	5.99	10.54	18.57	14.29	8.95	13.29
	search	2.08	2.12	2.25	1.91	1.87	1.98	3.04	2.93	5.81
	delete	2.01	2.69	5.06	2.51	4.55	8.02	5.84	2.76	5.35

Time Unit : *sec*

Table 3.18. Run time on ordered inputs using integer keys (version 1 code)

n	operation	AVL	RB-T	RB-B	WB	BBST	DSL	TRP	SKIP
10,000	insert	0.12	0.17	0.12	0.18	0.27	0.23	0.08	0.20
	search	0.05	0.03	0.03	0.07	0.05	0.07	0.05	0.12
	ins/del	0.18	0.32	0.20	0.35	0.57	0.42	0.17	0.20
	search	0.05	0.05	0.05	0.05	0.03	0.07	0.05	0.13
	delete	0.05	0.10	0.07	0.13	0.23	0.15	0.05	0.07
50,000	insert	0.75	1.02	0.92	1.25	1.10	0.98	0.47	0.92
	search	0.32	0.27	0.27	0.28	0.20	0.33	0.32	0.62
	ins/del	1.28	2.17	1.25	2.20	2.40	2.03	0.80	1.07
	search	0.28	0.28	0.27	0.28	0.20	0.30	0.37	0.62
	delete	0.30	0.75	0.37	0.85	1.05	0.65	0.30	0.27
100,000	insert	1.50	2.52	1.70	2.58	2.53	2.58	0.90	1.72
	search	0.70	0.60	0.57	0.70	0.42	0.70	0.63	1.23
	ins/del	2.60	4.68	2.53	4.78	5.13	4.42	1.52	2.43
	search	0.63	0.60	0.55	0.62	0.42	0.70	0.58	1.35
	delete	0.62	1.65	0.78	1.87	2.15	1.42	0.45	0.55
200,000	insert	3.12	4.82	3.38	5.67	5.25	4.72	1.80	3.52
	search	1.38	1.30	1.22	1.33	0.90	1.60	1.25	2.70
	ins/del	5.15	10.40	5.35	10.40	10.88	9.48	3.10	5.13
	search	1.33	1.33	1.18	1.32	0.90	1.50	1.28	2.72
	delete	1.35	3.63	1.68	4.12	4.58	2.98	0.93	1.12

Time Unit : *sec*

Table 3.19: Run time on random real inputs (version 1 code)

n	operation	BST	AVL	RB-T	RB-B	WB	BBST	DSL	TRP	SKIP
10,000	insert	0.14	0.15	0.21	0.17	0.23	0.36	0.22	0.23	0.30
	search	0.09	0.07	0.09	0.10	0.08	0.10	0.13	0.13	0.21
	ins/del	0.24	0.27	0.51	0.32	0.38	0.79	0.62	0.41	0.53
	search	0.09	0.08	0.09	0.10	0.08	0.10	0.12	0.12	0.21
	delete	0.09	0.09	0.17	0.14	0.14	0.30	0.28	0.11	0.19
50,000	insert	0.94	0.97	1.22	0.86	1.29	1.93	1.48	1.19	1.67
	search	0.64	0.52	0.50	0.51	0.51	0.52	0.87	0.71	1.44
	ins/del	1.68	1.77	2.74	1.53	2.29	4.22	3.93	2.17	3.15
	search	0.66	0.55	0.56	0.54	0.56	0.55	0.86	0.71	1.33
	delete	0.63	0.67	1.10	0.72	0.92	1.76	1.80	0.69	1.22
100,000	insert	2.06	1.85	2.34	1.90	2.66	4.36	3.05	2.67	3.61
	search	1.43	1.13	1.09	1.13	1.14	1.16	1.84	1.66	3.00
	ins/del	3.63	3.93	6.18	3.33	4.96	9.30	8.45	4.84	7.10
	search	1.45	1.26	1.27	1.17	1.26	1.22	1.83	1.65	3.01
	delete	1.39	1.50	2.51	1.55	2.03	3.95	3.91	1.61	2.75
200,000	insert	4.34	3.95	5.20	3.88	5.56	9.33	6.77	5.81	7.90
	search	3.19	2.49	2.42	2.50	2.45	2.57	4.14	3.67	6.62
	ins/del	8.01	8.25	13.78	7.29	10.65	20.46	18.88	10.48	15.83
	search	3.21	2.83	2.86	2.62	2.74	2.66	4.08	3.73	6.74
	delete	3.11	3.27	5.55	3.41	4.43	8.80	8.56	3.54	6.04

Time Unit : *sec*

Table 3.20. Run time on ordered real inputs (version 1 code)

n	operation	AVL	RB-T	RB-B	WB	BBST	DSL	TRP	SKIP
10,000	insert	0.13	0.22	0.15	0.25	0.20	0.25	0.12	0.30
	search	0.07	0.08	0.07	0.07	0.07	0.10	0.07	0.15
	ins/del	0.23	0.42	0.27	0.40	0.43	0.47	0.18	0.28
	search	0.07	0.05	0.08	0.08	0.05	0.08	0.08	0.12
	delete	0.07	0.17	0.08	0.15	0.20	0.20	0.05	0.07
50,000	insert	1.15	1.58	1.12	1.85	1.12	1.30	0.67	1.35
	search	0.42	0.42	0.43	0.40	0.30	0.53	0.38	0.82
	ins/del	1.28	2.75	1.57	2.57	2.37	3.02	0.92	1.40
	search	0.40	0.42	0.42	0.48	0.30	0.53	0.40	0.75
	delete	0.38	0.95	0.55	0.93	0.97	1.15	0.33	0.35
100,000	insert	1.77	3.23	2.12	3.35	2.77	3.13	1.17	2.42
	search	0.90	0.87	0.90	0.88	0.63	1.12	0.92	1.70
	ins/del	3.00	6.00	3.42	5.38	5.13	6.32	1.92	3.22
	search	0.97	0.92	0.88	0.98	0.63	1.12	0.82	1.70
	delete	0.87	2.08	1.17	2.05	2.10	2.40	0.70	0.67
200,000	insert	3.92	6.42	4.27	7.25	4.92	6.03	2.58	4.93
	search	1.92	1.87	1.92	1.88	1.32	2.40	1.85	3.87
	ins/del	5.78	13.80	7.33	11.88	10.93	13.72	3.75	6.67
	search	1.90	1.93	1.92	2.13	1.33	2.38	1.75	3.97
	delete	1.67	4.55	2.48	4.45	4.43	5.10	1.40	1.35

Time Unit : *sec*

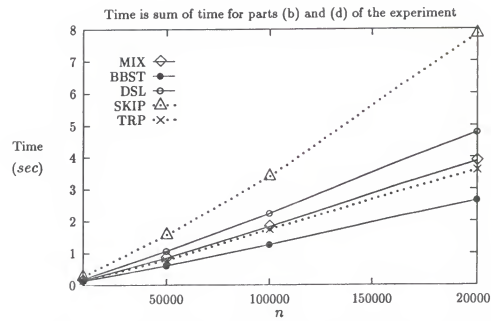


Figure 3.16. Run time on ordered real inputs (version 1 code)

lists performed best on part (a). Of the deterministic methods, BBSTs slightly outperformed the others on part (a). On parts (b) – (e), AVL, RB-T, RB-B, WB, and BBSTs were quite competitive and outperformed BSTs and the randomized schemes. BBSTs performed best on parts (b) and (d), RB-Ts did best on part (e) and RB-B and AVL did best on part (c). In comparing the results of Table 3.21 to those of Table 3.11 (using version 1 code), we see that the change to version 2 generally increased the comparison cost of the deterministic tree structures by about 25%. For the DSL, the change in code had mixed results. Notice that for RB-T and DSLs, the comparison count for parts (a), (c), and (e) are the same as for the version 1 code. This is because for inserts and deletes, it is necessary to do the equal check first when using these structures. For SKIPs the count is the same for all five parts as the version 1 and 2 codes are the same.

With ordered data (Table 3.22), treaps required the fewest comparisons for part (a). Skip lists did best on parts (c) and (e), and AVL trees generally outperformed the other structures on parts (b) and (d). Once again, the comparison counts were generally higher using the version 2 code than using the version 1 code.

Run time data using real keys is given in Tables 3.23 and 3.24. The sum of the run time for parts (b) and (d) of the experiment is graphed in Figure 3.17 for random data and in Figure 3.18 for ordered data. The graph of Figure 3.17 shows only one line MIX for AVL, RB-T, RB-B, WB, and BBST while that of Figure 3.18 shows MIX for AVL, RB-T, RB-B, and WB as the times for these are very close. With random data, RB-B generally performed best on part (a), on parts (b) and (d), the front runner varied among AVL, RB-T, and WB, and on parts (c) and (e) RB-Bs generally did best. On ordered data, TRPs did best on parts (a), (c), and (e) while BBSTs did best on parts (b) and (d).

3.8 Conclusion

We have developed a new weight balanced data structure called β-BBST. This was developed for the representation of a dictionary. In developing the insert/delete algorithms, we sought to minimize the search cost of the resulting tree. Our experimental results show that BBSTs generally have the best search cost of the structures considered. Furthermore, this translates into reduced search time when the key comparison cost is relatively high (e.g., for real keys). The insert and delete algorithms for β-BBSTs are not as efficient as those for other dictionary structures (such as AVL trees). As a result, we recommend β-BBSTs for environments where searches

Table 3.21: The number of key comparisons on random inputs (version 2 code)

n	operation	BST	AVL	RB-T	RB-B	WB	BBST	DSL	TRP	SKIP
	insert	332753	262198	262838	262726	263177	260896	276247	375698	224757
	search	322753	241557	242258	242262	242824	240126	348403	329411	255072
10,000	ins/del	650901	514371	515184	513920	515732	518124	923524	755629	519430
	search	318749	241536	247130	243191	242867	240126	335613	320612	256124
	delete	271004	206558	200218	206721	207622	207210	526242	300619	231745
	insert	1983939	1546988	1550701	1549795	1554520	1539666	1640660	2184066	1357076
	search	1933939	1443879	1447870	1446679	1452920	1435927	2043618	1921255	1537547
50,000	ins/del	3892221	3043090	3058045	3040654	3055092	3061443	5351715	4393520	2996512
	search	1913068	1443837	1476158	1451163	1452625	1435726	1969926	1909919	1501731
	delete	1674128	1267637	1242426	1268881	1275612	1270935	3077266	1815736	1373858
	insert	4245062	3297162	3305332	3302792	3314410	3281959	3513401	4637264	2919371
	search	4145062	3090057	3098143	3096011	3111095	3074661	4387427	4161175	3188621
100,000	ins/del	8336846	6490752	6564352	6486464	6520729	6528606	11545200	9484761	6399463
	search	4102672	3089826	3176862	3105465	3110184	3074305	4270168	4224698	3225343
	delete	3623179	2738267	2692672	2740846	2756006	2744369	6561272	4008111	2981173
	insert	9045367	6999791	7016676	7012317	7040203	6969465	7483199	9834444	6178596
	search	8845367	6584279	6603044	6599643	6633218	6554714	9373163	8752856	6697223
200,000	ins/del	17782478	13790643	13940982	13789492	13862467	13867876	24207106	19825904	13377747
	search	8757433	6585758	6747566	6618833	6630334	6554354	8995685	8889053	6680642
	delete	7800524	5882302	5800203	5889983	5923552	5893982	13811271	8456931	6149268

Table 3.22: The number of key comparisons on ordered inputs (version 2 code)

n	operation	AVL	RB-T	RB-B	WB	BBST	DSL	TRP	SKIP
	insert	267234	376228	442766	302034	261108	435199	216958	247129
	search	237262	239442	247706	237298	243110	372444	332060	256706
10,000	ins/del	493028	718040	727620	562808	558770	983676	482499	354566
	search	240910	238834	246330	238320	242736	349730	344982	250538
	delete	178076	276136	208216	204930	239028	468244	183080	84392
	insert	1568930	2233658	2672450	1791440	1545934	2585557	1375770	1422120
	search	1418962	1421560	1459588	1420858	1455936	2159176	1990474	1467217
50,000	ins/del	2881762	4311748	4360200	3301668	3251450	6019215	2742877	1973416
	search	1419154	1444824	1444258	1424442	1452494	2131862	1956194	1449810
	delete	1064956	1719212	1273918	1226262	1427504	2785792	1131612	486498
	insert	3337858	4767564	5744778	3824402	3301096	5521408	2898893	2925618
	search	3037892	3043084	3119128	3041676	3121098	4618272	4538718	2970715
100,000	ins/del	6113530	9223606	9320376	7027676	6930932	12788447	5492066	4406427
	search	3038276	3089612	3088470	3048844	3114012	4563600	4158994	3277089
	delete	2279908	3737982	2747792	2635270	3056908	5971196	2303622	961283
	insert	7075714	10135418	12289426	8131870	7006166	11743159	5756575	6403207
	search	6475750	6486128	6638204	6483310	6646168	9836456	9102954	6448304
200,000	ins/del	12927066	19647336	19840728	14904040	14671602	27076911	11290926	9062233
	search	6476518	6579184	6576890	6497646	6634260	9727066	8918638	6458321
	delete	4859812	8075474	5895538	5636096	6529928	12741948	4894044	1995215

Table 3.23: Run time on random real inputs (version 2 code)

n	operation	BST	AVL	RB-T	RB-B	WB	BBST	DSL	TRP	SKIP
10,000	insert	0.15	0.14	0.20	0.18	0.25	0.36	0.23	0.25	0.31
	search	0.10	0.08	0.10	0.11	0.09	0.11	0.13	0.16	0.21
	ins/del	0.27	0.27	0.52	0.34	0.47	0.80	0.64	0.50	0.54
	search	0.10	0.08	0.10	0.11	0.09	0.11	0.13	0.14	0.21
	delete	0.10	0.10	0.20	0.14	0.18	0.32	0.29	0.14	0.19
50,000	insert	1.02	0.98	1.15	0.89	1.46	1.88	1.44	1.34	1.65
	search	0.69	0.55	0.57	0.55	0.57	0.55	0.89	0.83	1.42
	ins/del	1.79	1.80	2.99	1.59	2.93	3.97	3.82	2.44	3.16
	search	0.71	0.60	0.63	0.55	0.57	0.56	0.87	0.79	1.32
	delete	0.67	0.67	1.22	0.66	1.19	1.63	1.80	0.75	1.21
100,000	insert	2.15	2.00	2.58	1.90	3.18	4.01	3.11	2.95	3.69
	search	1.52	1.21	1.24	1.18	1.23	1.23	1.97	1.84	3.04
	ins/del	3.88	3.92	6.74	3.46	6.28	8.73	8.50	5.39	7.18
	search	1.55	1.32	1.45	1.25	1.29	1.27	1.95	1.82	2.98
	delete	1.51	1.49	2.75	1.45	2.57	3.64	3.93	1.73	2.77
200,000	insert	5.04	4.45	5.79	4.28	6.92	9.20	7.05	6.81	8.01
	search	3.43	2.63	2.70	2.64	2.73	2.69	4.43	4.00	6.60
	ins/del	8.92	8.87	15.36	7.88	13.85	19.53	19.55	12.17	16.11
	search	3.43	2.98	3.13	2.73	2.83	2.77	4.37	4.02	6.70
	delete	3.33	3.32	6.08	3.20	5.65	8.24	8.91	3.88	6.04

Time Unit : *sec*

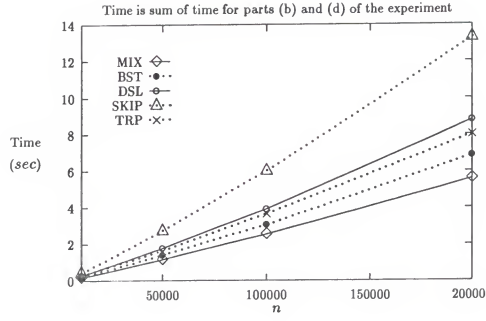


Figure 3.17. Run time on random real inputs (version 2 code)

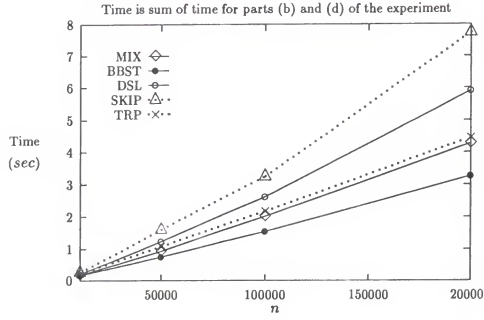


Figure 3.18. Run time on ordered real inputs (version 2 code)

Table 3.24. Run time on ordered real inputs (version 2 code)

n	operation	AVL	RB-T	RB-B	WB	BBST	DSL	TRP	SKIP
10,000	insert	0.17	0.23	0.28	0.27	0.30	0.23	0.15	0.30
	search	0.08	0.08	0.12	0.08	0.08	0.12	0.12	0.13
	ins/del	0.23	0.43	0.40	0.47	0.60	0.48	0.17	0.27
	search	0.08	0.08	0.07	0.08	0.08	0.08	0.10	0.13
	delete	0.08	0.15	0.12	0.17	0.20	0.20	0.08	0.05
50,000	insert	0.83	1.45	1.43	1.57	1.37	1.35	0.82	1.18
	search	0.45	0.48	0.48	0.47	0.38	0.60	0.50	0.83
	ins/del	1.35	2.65	1.95	2.75	2.47	3.05	1.05	1.42
	search	0.45	0.47	0.45	0.47	0.37	0.63	0.58	0.77
	delete	0.45	1.05	0.50	1.00	1.03	1.17	0.43	0.33
100,000	insert	1.78	2.75	2.73	3.43	2.63	3.23	1.33	2.18
	search	0.97	0.98	1.00	1.03	0.77	1.30	1.15	1.55
	ins/del	2.85	6.22	3.98	6.00	5.33	6.37	2.02	3.33
	search	0.97	1.10	0.98	1.02	0.77	1.32	1.03	1.70
	delete	0.97	2.18	1.05	2.15	2.22	2.43	0.63	0.67
200,000	insert	3.78	6.08	5.43	7.18	5.37	6.07	2.87	5.23
	search	2.08	2.13	2.13	2.17	1.63	3.10	2.27	3.47
	ins/del	6.13	13.93	8.48	13.42	11.33	13.60	4.10	7.02
	search	2.12	2.15	2.13	2.17	1.63	2.80	2.18	4.27
	delete	2.03	4.75	2.27	4.77	4.72	5.18	1.35	1.35

Time Unit : *sec*

are done with much greater frequency than inserts and/or deletes. Based on our experiments, we conclude that AVL trees remain the best dictionary structure for general applications.

We have also proposed two simplified versions of the BBST called SBBST and BBSTD. The SBBST seeks only to provide logarithmic run time per operation and unlike the general BBST, does not reduce search cost at every opportunity. The SBBST provides slightly better balance than provided by WB(α) trees. The BBSTD does not attempt to maintain β-balance. However it performs rotations to reduce search cost whenever possible. Both versions are very competitive with BBSTs. The SBBST exhibited much better run time performance than BBSTs on random data and the BBSTD slightly outperformed the BBST on ordered data. However, BBSTs generated trees with the lowest search cost (though not by much).

CHAPTER 4
WEIGHT BIASED LEFTIST TREES AND MODIFIED SKIP LISTS

4.1 Introduction

Several data structures (e.g., heaps, leftist trees [9], binomial heaps [10]) have been proposed for the representation of a (single ended) priority queue. Heaps permit one to delete the min element and insert an arbitrary element into an n element priority queue in $O(\log n)$ time. Leftist trees support both these operations and the merging of pairs of priority queues in logarithmic time. Using binomial heaps, inserts and combines take $O(1)$ time and a delete-min takes $O(\log n)$ amortized time. In this chapter, we begin in Section 4.2, by developing the weight biased leftist tree. This is similar to a leftist tree. However biasing of left and right subtrees is done by number of nodes rather than by length of paths. Experimental results presented in Section 4.5 show that weight biased leftist trees provide better performance than provided by leftist trees. The experimental comparisons of Section 4.5 also include a comparison with heaps and binomial heaps as well as with unbalanced binary search trees and the probabilistic structures treap [1] and skip lists [26].

In Section 4.3, we propose a fixed node size representation for skip lists. The new structure is called modified skip lists and is experimentally compared with the

variable node size structure skip lists. Our experiments indicate that modified skip lists are faster than skip lists when used to represent dictionaries.

Modified skip lists are augmented by a thread in Section 4.4 to obtain a structure suitable for use as a priority queue. For completeness, we include, in Section 4.5, a comparison of data structures for double ended priority queues.

4.2 Weight Biased Leftist Trees

Let T be an extended binary tree. For any internal node x of T, let $LeftChild(x)$ and $RightChild(x)$, respectively, denote the left and right children of x. The weight, $w(x)$, of any node x is the number of internal nodes in the subtree with root x. The length, $shortest(x)$, of a shortest path from x to an external node satisfies the recurrence

$$shortest(x) = \begin{cases} 0 \text{ if } x \text{ is an external node} \\ 1 + min\{shortest(LeftChild(x)), shortest(RightChild(x))\} \text{ otherwise.} \end{cases}$$

Definition [9] A *leftist tree* (LT) is a binary tree such that if it is not empty, then

$$shortest(LeftChild(x)) \geq shortest(RightChild(x))$$

for every internal node x.

A weight biased leftist tree (WBLT) is defined by using the weight measure in place of the measure *shortest*.

Definition A *weight biased leftist tree* (WBLT) is a binary tree such that if it is not empty, then

$$weight(LeftChild(x)) \geq weight(RightChild(x))$$

for every internal node x.

It is known [9] that the length, $rightmost(x)$, of the rightmost root to external node path of any subtree, x, of a leftist tree satisfies

$$rightmost(x) \leq \log_2(w(x)+1).$$

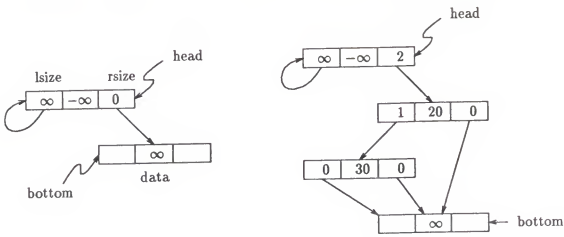
The same is true for weight biased leftist trees.

Theorem 13 *Let x be any internal node of a weight biased leftist tree.* $rightmost(x) \leq \log_2(w(x)+1)$.

Proof The proof is by induction on $w(x)$. When $w(x) = 1$, $rightmost(x) = 1$ and $\log_2(w(x)+1) = \log_2 2 = 1$. For the induction hypothesis, assume that $rightmost(x) \leq \log_2(w(x)+1)$ whenever $w(x) < n$. When $w(x) = n$, $w(RightChild(x)) \leq (n-1)/2$ and $rightmost(x) = 1 + rightmost(RightChild(x)) \leq 1 + \log_2((n-1)/2 + 1) = 1 + \log_2(n+1) - 1 = \log_2(n+1)$. □

Definition A min (max)-WBLT is a WBLT that is also a min (max) tree.

Each node of a min-WBLT has the fields: *lsize* (number of internal nodes in left subtree), *rsize*, *left* (pointer to left subtree), *right*, and *data*. While the



(a) Empty min-WBLT (b) Nonempty min-WBLT

Figure 4.1. Example min-WBLTs

number of size fields in a node may be reduced to one, two fields result in a faster implementation. We assume a head node *head* with $lsize = \infty$ and $lchild = head$. In addition, a bottom node *bottom* with $data.key = \infty$. All pointers that would normally be *nil* are replaced by a pointer to *bottom*. Figure 4.1(a) shows the representation of an empty min-WBLT and Figure 4.1(b) shows an example non empty min-WBLT. Notice that all elements are in the right subtree of the head node.

Min (max)-WBLTs can be used as priority queues in the same way as min (max)-LTs. For instance, a min-WBLT supports the standard priority queue operations of insert and delete-min in logarithmic time. In addition, the combine operation (i.e., join two priority queues together) can also be done in logarithmic time. The algorithms for these operations have the same flavor as the corresponding ones for min-LTs. A high level description of the insert and delete-min algorithm for min-WBLT is given in Figures 4.2 and 4.3, respectively. The algorithm to combine two

```
procedure Insert(d) ;
{insert d into a min-WBLT}
begin
create a node x with x.data = d ;
t = head ; {head node}
while (t.right.data.key < d.key) do
    begin
    t.rsize = t.rsize + 1 ;
    if (t.lsize < t.rsize) then
        begin swap t's children ; t = t.left ; end
    else t = t.right ;
    end ;
x.left = t.right ; x.right = bottom ;
x.lsize = t.rsize ; x.rsize = 0 ;
if (t.lsize = t.rsize) then {swap children}
    begin
    t.right = t.left ;
    t.left = x ; t.lsize = x.lsize + 1 ;
    end
else
    begin t.right = x ; t.rsize = t.rsize + 1 ; end ;
end ;
```

Figure 4.2. min-WBLT Insert

min-WBLTs is similar to the delete-min algorithm. The time required to perform each of the operations on a min-WBLT T is $O(rightmost(T))$.

Notice that while the insert and delete-min operations for min-LTs require a top-down pass followed by a bottom-up pass, these operations can be performed by a single top-down pass in min-WBLTs. Hence, we expect min-WBLTs to outperform min-LTs.

4.3 Modified Skip Lists

Skip lists were proposed in Pugh [26] as a probabilistic solution for the dictionary problem (i.e., represent a set of keys and support the operations of search, insert, and delete). The essential idea in skip lists is to maintain upto *lmax* ordered chains

```
procedure Delete-min ;
begin
x = head.right ;
if (x = bottom) then return ; {empty tree}
head.right = x.left ; head.rsize = x.lsize ;
a = head;
b = x.right ; bsize = x.rsize ;
delete x ;
if (b = bottom) then return ;
r = a.right ;
while (r ≠ bottom) do
    begin
    s = bsize + a.rsize ; t = a.rsize ;
    if (a.lsize < s) then  {work on a.left}
        begin
        a.right = a.left ; a.rsize = a.lsize ; a.lsize = s ;
        if (r.data.key > b.data.key) then
            begin a.left = b ; a = b ; b = r ; bsize = t ; end
        else
            begin a.left = r ; a = r ; end
        end
    else
        do symmetric operations on a.right ;
    r = a.right ;
    end ;
if (a.lsize < bsize) then
    begin
    a.right = a.left ; a.left = b ;
    a.rsize = a.lsize ; a.lsize = bsize ;
    end
else
    begin a.right = b ; a.rsize = bsize ; end ;
end ;
```

Figure 4.3. min-WBLT Delete-min

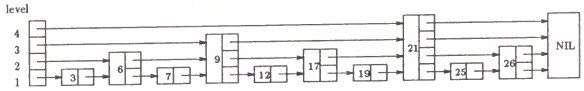


Figure 4.4. Skip Lists

designated as level 1 chain, level 2 chain, etc. If we currently have *lcurrent* number of chains, then all n elements of the dictionary are in the level 1 chain and for each l, $2 \leq l \leq lcurrent$, approximately a fraction p of the elements on the level $l-1$ chain are also on the level l chain. Ideally, if the level $l-1$ chain has m elements then the approximately $m \times p$ elements on the level l chain are about $1/p$ apart in the level $l-1$ chain. Figure 4.4 shows an ideal situation for the case $lcurrent = 4$ and $p = 1/2$.

While the search, insert, and delete algorithms for skip lists are simple and have probabilistic complexity $O(\log n)$ when the level 1 chain has n elements, skip lists suffer from the following implementational drawbacks:

1. In programming languages such as Pascal, it isn't possible to have variable size nodes. As a result, each node has one *data* field, and *lmax* pointer fields. So, the n element nodes have a total of $n \times lmax$ pointer fields even though only about $n/(1-p)$ pointers are necessary. Since *lmax* is generally much larger than 3 (the recommended value is $\log_{1/p} nMax$ where $nMax$ is the largest number of elements expected in the dictionary), skip lists require more space than WBLTs.

2. While languages such as C and C++ support variable size nodes and we can construct variable size nodes using simulated pointers [27] in languages such as Pascal that do not support variable size nodes, the use of variable size nodes requires more complex storage management techniques than required by the use of fixed size nodes. So, greater efficiency can be achieved using simulated pointers and fixed size nodes.

With these two observations in mind, we propose a modified skip list (MSL) structure in which each node has one *data* field and three pointer fields: *left*, *right*, and *down*. Notice that this means MSLs use four fields per node while WBLTs use five (as indicated earlier this can be reduced to four at the expense of increased run time). The *left* and *right* fields are used to maintain each level l chain as a doubly linked list and the *down* field of a level l node x points to the leftmost node in the level $l-1$ chain that has key value larger than the key in x. Figure 4.5 shows the modified skip list that corresponds to the skip list of Figure 4.4. Notice that each element is in exactly one doubly linked list. We can reduce the number of pointers in each node to two by eliminating the field *left* and having *down* point one node the left of where it currently points (except for head nodes whose down fields still point to the head node of the next chain). However, this results in a less time efficient implementation. H and T, respectively, point to the head and tail of the level *lcurrent* chain.

A high level description of the algorithms to search, insert, and delete are given in Figures 4.6, 4.7, and 4.8. The next theorem shows that their probabilistic complexity is $O(\log n)$ where n is the total number of elements in the dictionary.

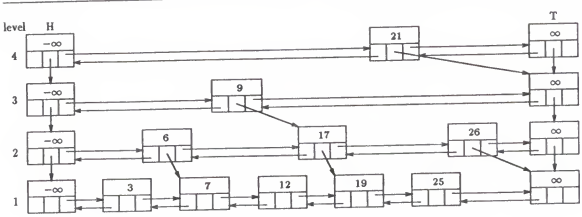


Figure 4.5. Modified Skip Lists

```
procedure Search(key) ;
begin
p = H ;
while (p ≠ nil) do
    begin
    while (p.data.key < key) do
        p = p.right ;
    if (p.data.key = key) then report and stop
    else p = p.left.down ; {1 level down}
    end ;
end ;
```

Figure 4.6. MSL Search

```
procedure Insert(d) ;
begin
randomly generate the level k at which d is to be inserted ;
search the MSL H for d.key saving information useful for insertion ;
if d.key is found then fail ; {duplicate}
get a new node x and set x.data = d ;
if ((k > lcurrent) and (lcurrent ≠ lmax)) then
    begin
    lcurrent = lcurrent + 1 ;
    create a new chain with a head node, node x, and a tail and
    connect this chain to H ;
    update H ;
    set x.down to the appropriate node in the level lcurrent − 1 chain
    (to nil if k = 1) ;
    end
else
    begin
    insert x into the level k chain ;
    set x.down to the appropriate node in the level k − 1 chain
    (to nil if k = 1) ;
    update the down field of nodes on the level k + 1 chain (if any) as needed ;
    end ;
end ;
```

Figure 4.7. MSL Insert

```
procedure Delete(z) ;
begin
search the MSL H for a node x with data.key = z saving information useful
for deletion;
if not found then fail ;
let k be the level at which z is found ;
for each node p on level k + 1 that has p.down = x, set p.down = x.right ;
delete x from the level k list ;
if the list at level lcurrent becomes empty then
    delete this and succeeding empty lists until we reach the first non empty list,
    update lcurrent ;
end ;
```

Figure 4.8. MSL Delete

Theorem 14 *The probabilistic complexity of the MSL operations is $O(\log n)$.*

Proof We establish this by showing that our algorithms do at most a logarithmic amount of additional work than do those of Pugh [26]. Since the algorithms of Pugh [26] has probabilistic $O(\log n)$ complexity, so also do ours. During a search, the extra work results from moving back one node on each level and then moving down one level. When this is done from any level other than *lcurrent*, we expect to examine upto $c = 1/p - 1$ additional nodes on the next lower level. Hence, upto $c(lcurrent - 2)$ additional nodes get examined. During an insert, we also need to verify that the element being inserted isn't one of the elements already in the MSL. This requires an additional comparison at each level. So, MSLs may make upto $c(lcurrent - 2) + lcurrent$ additional compares during an insert. The number of *down* pointers that need to be changed during an insert or delete is expected to be $\sum_{i=1}^{\infty} ip^i = \frac{1}{(1-p)^2}$. Since c and p are constants and $lmax = \log_{1/p} n$, the expected additional work is $O(\log n)$. □

The relative performance of skip lists and modified skip lists as a data structure for dictionaries was determined by programming the two in C. Both were implemented using simulated pointers. The simulated pointer implementation of skip lists used fixed size nodes. This avoided the use of complex storage management methods and biased the run time measurements in favor of skip lists. For the case of skip lists, we used $p = 1/4$ and for MSLs, $p = 1/5$. These values of p were found to work best for each structure. *lmax* was set to 16 for both structures.

We experimented with n = 10,000, 50,000, 100,000, and 200,000. For each n, the following five part experiment was conducted:

(a) start with an empty structure and perform n inserts;

(b) search for each item in the resulting structure once; items are searched for in the order they were inserted

(c) perform an alternating sequence of n inserts and n deletes; in this, the n elements inserted in (a) are deleted in the order they were inserted and n new elements are inserted

(d) search for each of the remaining n elements in the order they were inserted

(e) delete the n elements in the order they were inserted.

For each n, the above five part experiment was repeated ten times using different random permutations of distinct elements. For each sequence, we measured the total number of element comparisons performed and then averaged these over the ten sequences. The average number of comparisons for each of the five parts of the experiment are given in Table 4.1.

Also given in this table is the number of comparisons using ordered data. For this data set, elements were inserted and deleted in the order $1, 2, 3, \ldots$. For the case of random data, MSLs make 40% to 50% more comparisons on each of the five parts of the experiment. On ordered inputs, the disparity is even greater with MSLs making 30% to 140% more comparison. Table 4.2 gives the number of levels in SKIP and MSL. The first number of each entry is the number of levels following part (a) of the experiment and the second the number of levels following part (b). As can be

Table 4.1. The number of key comparisons

n	operation	random inputs		ordered inputs	
		SKIP	MSL	SKIP	MSL
10,000	insert	224757	322499	247129	318854
	search	255072	362865	256706	339019
	ins/del	519430	734161	354566	560219
	search	256124	349591	250538	339121
	delete	231745	320594	84392	185489
50,000	insert	1357076	1950583	1422120	1911818
	search	1537547	1965649	1467217	1836713
	ins/del	2996512	4142186	1973416	3204400
	search	1501731	2038774	1449810	1989550
	delete	1373858	1853671	486498	931975
100,000	insert	2919371	4146428	2925618	4275880
	search	3188621	4315576	2970715	4082193
	ins/del	6399463	9103135	4406427	6895510
	search	3225343	4427979	3277089	4345874
	delete	2981173	4161994	961283	2052638
200,000	insert	6178596	8927523	6403207	9022631
	search	6697223	9273707	6448304	8946474
	ins/del	13377747	19370831	9054078	9062233
	search	6680642	9662006	6458321	9197714
	delete	6149268	9101721	1995215	4837867

Table 4.2. Number of levels

n	random inputs		ordered inputs	
	SKIP	MSL	SKIP	MSL
10,000	8,8	7,7	8,8	7,7
50,000	9,9	7,7	9,9	7,7
100,000	9,9	7,8	9,9	7,8
200,000	9,9	8,9	9,9	8,9

seen, the number of levels is very comparable for both structures. MSLs generally had one or two levels fewer than SKIPs had.

Despite the large disparity in number of comparisons, MSLs generally required less time than required by SKIPs (see Table 4.3 and Figure 4.9). Integer keys were used for our run time measurements. In many practical situations the observed time difference will be noticeably greater as one would need to code skip lists using more complex storage management techniques to allow for variable size nodes.

4.4 MSLs As Priority Queues

At first glance, it might appear that skip lists are clearly a better choice than modified skip lists for use as a priority queue. The min element in a skip list is the first element in the level one chain. So, it can be identified in $O(1)$ time and then deleted in $O(\log n)$ probabilistic time. In the case of MSLs, the min element is the first one in one of the *lcurrent* chains. This can be identified in logarithmic time using a loser tree whose elements are the first element from each MSL chain. By using an additional pointer field in each node, we can thread the elements in an MSL into a chain. The elements appear in non-decending order on this chain. The

Table 4.3. Run time

n	operation	random inputs		ordered inputs	
		SKIP	MSL	SKIP	MSL
10,000	insert	0.24	0.18	0.20	0.17
	search	0.18	0.12	0.12	0.07
	ins/del	0.45	0.35	0.20	0.20
	search	0.18	0.12	0.13	0.07
	delete	0.16	0.12	0.07	0.05
50,000	insert	1.36	1.22	0.92	0.80
	search	1.25	0.98	0.62	0.38
	ins/del	2.73	2.53	1.07	1.08
	search	1.16	1.00	0.62	0.42
	delete	1.10	0.83	0.27	0.23
100,000	insert	2.84	2.86	1.72	1.60
	search	2.63	2.39	1.23	0.85
	ins/del	6.13	5.80	2.43	2.28
	search	2.61	2.33	1.35	0.92
	delete	2.41	2.02	0.55	0.52
200,000	insert	6.25	6.49	3.52	3.47
	search	5.85	5.34	2.70	1.87
	ins/del	13.29	13.02	5.13	4.75
	search	5.81	5.51	2.72	1.92
	delete	5.35	4.85	1.12	1.18

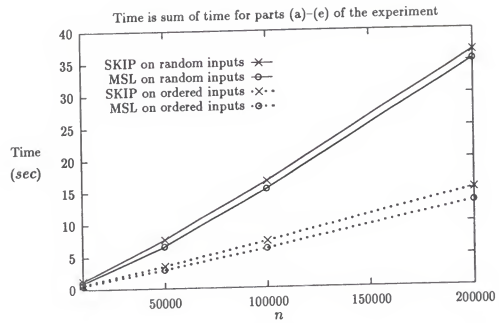


Figure 4.9. Run time

resulting threaded structure is referred to as TMSL (threaded modified skip lists). A delete min operation can be done in $O(1)$ expected time when a TMSL is used. The expected time for an insert remains $O(\log n)$. The algorithms for the insert and delete min operations for TMSLs are given in Figures 4.10 and 4.11, respectively. The last step of Figure 4.10 is implemented by first finding the largest element on level 1 with key $< d.key$ (for this, start at level $lcurrent - 1$) and then follow the threaded chain.

Theorem 15 *The expected complexity of an insert and delete-min operation in a TMSL is $O(\log n)$ and $O(1)$, respectively.*

Proof Follows from the notion of a thread, Theorem 14, and Pugh [26]. □

```
procedure Insert(d) ;
begin
randomly generate the level k at which d is to be inserted ;
get a new node x and set x.data = d ;
if ((k > lcurrent) and (lcurrent ≠ lmax)) then
    begin
    lcurrent = lcurrent + 1 ;
    create a new chain with a head node, node x, and a tail and
    connect this chain to H ;
    update H ;
    set x.down to the appropriate node in the level lcurrent − 1 chain
    (to nil if k = 1) ;
    end
else
    begin
    insert x into the level k chain ;
    set x.down to the appropriate node in the level k − 1 chain (to nil if k = 1) ;
    update the down field of nodes on the level k + 1 chain (if any) as needed ;
    end ;
find node with largest key < d.key and insert x into threaded list ;
end ;
```

Figure 4.10. TMSL Insert

```
procedure Delete-min ;
begin
delete the first node x from the thread list ;
let k be the level x is on ;
delete x from the level k list (note there are no down fields on level k + 1
that need to be updated) ;
if the list at level lcurrent becomes empty then
    delete this and succeeding empty lists until we reach the first non empty list,
    update lcurrent ;
end ;
```

Figure 4.11. TMSL Delete-min

```
procedure Delete-max ;
begin
delete the last node x from the thread list ;
let k be the level x is on ;
delete x from the level k list updating p.down for nodes on level k+1 as necessary;
if the list at level lcurrent becomes empty then
    delete this and succeeding empty lists until we reach the first non empty list,
    update lcurrent ;
end ;
```

Figure 4.12. TMSL Delete-max

TMSLs may be further extended by making the threaded chain a doubly linked list. This permits both delete-min and delete-max to be done in $\Theta(1)$ expected time and insert in $O(\log n)$ expected time. With this extension, TMSLs may be used to represent double ended priority queues.

4.5 Experimental Results For Priority Queues

The single-ended priority queue structures min heap (Heap), binomial heap (B-Heap), leftist trees (LT), weight biased leftist trees (WBLT), and TMSLs were programmed in C. In addition, priority queue versions of unbalanced binary search trees (BST), AVL trees, treaps (TRP), and skip lists (SKIP) were also programmed. The priority queue version of these structures differed from their normal dictionary versions in that the delete operation was customized to support only a delete min. For skip lists and TMSLs, the level allocation probability p was set to 1/4. While BSTs are normally defined only for the case when the keys are distinct, they are easily extended to handle multiple elements with the same key. In our extension, if a node has key x, then its left subtree has values $< x$ and its right values $\geq x$. To minimize the effects of system call overheads, all structures (other than Heap)

were programmed using simulated pointers. The min heap was programmed using a one-dimensional array.

For our experiments, we began with structures initialized with n = 100, 1,000, 100,000, and 100,000 elements and then performed a random sequence of 100,000 operations. This random sequence consists of approximately 50% insert and 50% delete min operations. The results are given in Tables 4.4, 4.5, and 4.6. In the data sets 'random1' and 'random2', the elements to be inserted were randomly generated while in the data set 'increasing' an ascending sequence of elements was inserted and in the data set 'decreasing', a descending sequence of elements was used. Since BST have very poor performance on the last two data sets, we excluded it from this part of the experiment. In the case of both random1 and random2, ten random sequences were used and the average of these ten is reported. The random1 and random2 sequences differed in that for random1, the keys were integers in the range $0..(10^6-1)$ while for random2, they were in the range 0..999. So, random2 is expected to have many more duplicates.

Table 4.4 gives the total number of comparisons made by each of the methods. On the two random data tests, weight biased leftist trees required the fewest number of comparisons except when $n = 100,000$. In this case, AVL trees required the fewest. With ascending data, treaps did best and with descending data, LTs and WBLTs did best. For both, each insert could be done with one comparison as both structures build a left skewed tree.

Table 4.4: The number of key comparisons

inputs	n	BST	Heap	B-Heap	LT	WBLT	TRP	SKIP	TMSL	AVL
random1	100	621307	823685	268224	165771	165525	407326	270584	503567	373683
	1,000	677570	1317728	383285	203550	202274	537418	397181	729274	542789
	10,000	726875	1693955	645664	476534	468713	757946	711869	1104805	685422
	100,000	1067670	1746376	1153516	1199207	1171181	1119554	1327083	1778579	877131
random2	100	522728	781808	260209	164288	164031	384721	261169	485995	358885
	1,000	612630	1273828	389333	199886	198720	518022	386837	707997	534406
	10,000	1027346	1576921	642968	448379	439014	768783	710612	1131421	685926
	100,000	5641638	1713676	1146214	1032043	978785	1746732	1339410	1800404	890190
increasing	100	–	564332	410119	552081	536796	363045	629085	836223	382513
	1,000	–	946659	655107	917664	882490	496223	1018473	1331317	584422
	10,000	–	1284712	814412	1234622	1192325	645765	1313298	1685844	747581
	100,000	–	1617645	923048	1550741	1560866	723825	1568819	1939835	902437
decreasing	100	–	836361	202402	50010	50010	362723	194245	425238	334965
	1,000	–	1394286	313587	50010	50010	515579	300298	637341	512942
	10,000	–	1950062	413840	50010	50010	558222	400082	836879	672910
	100,000	–	2400032	534821	50010	50010	648730	450090	936986	835044

n = the number of elements in initial data structures
Total number of operations performed = 100,000

The structure height initially and following the 100,000 operations is given in Table 4.5 for BSTs, Heaps, TRPs and AVL trees. For B-Heaps, the height of the tallest tree is given. For SKIPs and TMSLs, this table gives the number of levels. In the case of LT and WBLT, this table gives the length of the rightmost path following initialization and the average of its length following each of the 100,000 operations. The two leftist structures are able to maintain their rightmost paths so as to have a length much less than $\log_2(n+1)$.

The measured run times on a Sun Sparc 5 are given in Table 4.6. For this, the codes were compiled using the cc compiler in optimized mode. The run time for the data set random1 is graphed in Figure 4.13. The run time for the data set random2 and Heap, LT, WBLT, SKIP, TMSL, and AVL is graphed in Figure 4.14. For the data sets random1 and random2 with $n = 100$ and 1,000, WBLTs required least time. For random1 with $n = 10,000$, BSTs took least time while when $n = 100,000$, both BSTs and Heaps took least time. For random2 with $n = 10,000$, WBLTs were fastest while for $n = 100,000$, Heap was best. On the ordered data sets, BSTs have a very high complexity and are the poorest performers (times not shown in Table 4.6). For increasing data, Heap was best for $n = 100$, 1,000 and 10,000 and both Heap and TRP best for $n = 100,000$. For decreasing data, WBLTs were generally best. On all data sets, WBLTs always did at least as well (and often better) as LTs. Between SKIP and TMSL, we see that SKIP generally did better for small n and TMSL for large n.

Table 4.5: Height/level of the structures

inputs	n	BST	Heap	B-Heap	LT	WBLT	TRP	SKIP	TMSL	AVL
random1	100	13,13	7,6	1,6	4,2	4,2	12,11	4,4	4,4	8,7
	1,000	22,22	10,10	1,10	7,2	7,2	22,24	6,6	6,6	12,12
	10,000	31,32	14,14	1,14	8,4	8,4	33,30	8,7	8,7	16,16
	100,000	40,41	17,17	1,17	10,9	10,9	42,42	9,9	9,9	20,20
random2	100	13,16	7,7	1,7	4,2	4,2	14,17	4,4	4,4	8,7
	1,000	23,69	10,10	1,10	5,2	5,2	23,63	6,5	6,5	12,11
	10,000	39,93	14,14	1,14	6,4	6,4	36,83	8,7	8,7	16,15
	100,000	147,201	17,17	1,17	6,8	6,8	133,183	9,9	9,9	19,19
increasing	100	–	7,8	1,8	6,4	6,4	11,15	4,5	4,5	7,8
	1,000	–	10,11	1,11	9,7	9,7	24,24	6,6	6,6	10,11
	10,000	–	14,14	1,14	13,9	13,9	33,34	8,8	8,8	14,14
	100,000	–	17,17	1,17	16,11	16,11	46,41	9,9	9,9	17,17
decreasing	100	–	7,7	1,7	1,1	1,1	11,15	4,4	4,4	7,7
	1,000	–	10,10	1,10	1,1	1,1	24,24	6,6	6,6	10,10
	10,000	–	14,14	1,14	1,1	1,1	33,33	8,8	8,8	14,14
	100,000	–	17,17	1,17	1,1	1,1	46,46	9,9	9,9	17,17

n = the number of elements in initial data structures
Total number of operations performed = 100,000

Table 4.6: Run time using integer keys

inputs	n	BST	Heap	B-Heap	LT	WBLT	TRP	SKIP	TMSL	AVL
random1	100	0.32	0.32	0.53	0.30	0.23	0.56	0.36	0.37	0.50
	1,000	0.34	0.44	0.59	0.29	0.25	0.56	0.35	0.39	0.56
	10,000	0.38	0.57	0.98	0.62	0.49	0.71	0.62	0.59	0.70
	100,000	0.66	0.66	1.90	1.77	1.26	1.26	1.33	1.32	0.96
random2	100	0.29	0.30	0.55	0.27	0.25	0.55	0.32	0.35	0.50
	1,000	0.32	0.44	0.58	0.27	0.23	0.53	0.32	0.36	0.53
	10,000	0.49	0.51	0.93	0.57	0.44	0.70	0.59	0.57	0.67
	100,000	3.83	0.68	1.92	1.44	0.99	1.70	1.32	1.34	1.02
increasing	100	–	0.22	0.63	0.50	0.40	0.42	0.43	0.42	0.62
	1,000	–	0.35	0.92	0.95	0.70	0.48	0.78	0.58	0.58
	10,000	–	0.47	1.05	1.25	0.95	0.55	0.72	0.60	0.63
	100,000	–	0.60	1.30	1.83	1.40	0.60	0.83	0.65	0.78
decreasing	100	–	0.30	0.38	0.13	0.12	0.45	0.23	0.28	0.40
	1,000	–	0.45	0.47	0.13	0.10	0.50	0.28	0.33	0.48
	10,000	–	0.58	0.55	0.12	0.12	0.52	0.32	0.38	0.67
	100,000	–	0.70	0.73	0.12	0.12	0.55	0.35	0.45	0.80

Time Unit : *sec*
n = the number of elements in initial data structures
Total number of operations performed = 100,000

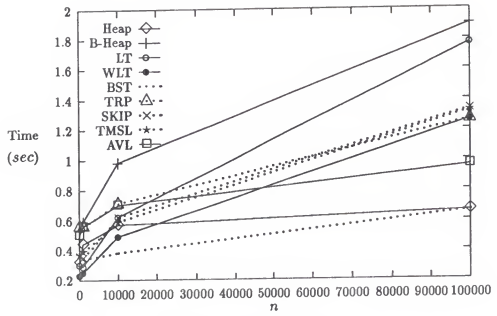


Figure 4.13. Run time on random1

Another way to interpret the time results is in terms of the ratio m/n (m = number of operations). In the experiments reported in Table 4.6, $m = 100,000$. As m/n increases, WBLTs and LTs perform better relative to the remaining structures. This is because as m increases, the (weight biased) leftist trees constructed are very highly skewed to the left and the length of the rightmost path is close to one.

Tables 4.7, 4.8, and 4.9 provide an experimental comparison of BSTs, AVL trees, MMHs (min-max heaps) [2], Deaps [7], TRPs, SKIPs, and TMSLs as a data structure for double ended priority queues. The experimental setup is similar to that used for single ended priority queues. However, this time the operation mix was 50% insert, 25% delete-min, and 25% delete-max. On the comparison measure, treaps did best on increasing data (except when $n = 100$) and skip lists did best when decreasing data was used. On all other data, AVL trees did best. As far as run time

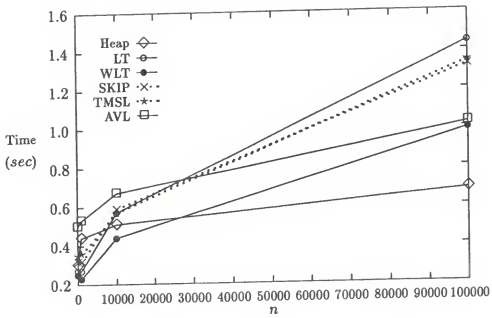


Figure 4.14. Run time on random2

is concerned, BSTs did best on the random data tests except when $n = 100,000$ and the set random2 was used. In this case, deaps and AVL trees took least time. For increasing data, treaps were best and for decreasing data, skip lists were best. The run time for the data set random1 is graphed in Figure 4.15. The run time for the data set random2 and MMH, Deap, SKIP, TMSL, and AVL is graphed in Figure 4.16.

4.6 Conclusion

We have developed two new data structures: weight biased leftist trees and modified skip lists. Experiments indicate that WBLTs have better performance (i.e., run time characteristic and number of comparisons) than LTs as a data structure for single ended priority queues and MSLs have a better performance than skip lists as

Table 4.7: The number of key comparisons

inputs	n	BST	MMH	Deap	TRP	SKIP	TMSL	AVL
random1	100	534197	994374	581071	402987	462690	666845	371435
	1,000	676634	1677964	912282	550363	698132	996774	545600
	10,000	738100	2328247	1122599	759755	1034669	1437925	693935
	100,000	1068250	2795369	1123423	1127616	1439387	1862864	878709
random2	100	514680	941925	557549	396910	447909	650533	357556
	1,000	908437	1651830	909747	564414	658284	949230	537616
	10,000	1339444	2239519	1145724	881889	1003300	1379690	689415
	100,000	5956407	2754760	1125210	1873922	1454138	1875706	891332
increasing	100	–	926017	592503	364026	624430	803024	363399
	1,000	–	1660945	1052894	507812	999886	1304223	562873
	10,000	–	2392627	1411640	614534	1373332	1766792	726087
	100,000	–	3015425	1849318	679120	1487465	1866498	878978
decreasing	100	–	926041	615944	360030	193021	413284	355999
	1,000	–	1711076	1062010	490022	287887	603089	563085
	10,000	–	2400014	1480876	676190	352780	732673	725079
	100,000	–	3035292	1851845	698740	450090	927278	878244

n = the number of elements in initial data structures
Total number of operations performed = 100,000

Table 4.8. Height/level of the structures

inputs	n	BST	MMH	Deap	TRP	SKIP	TMSL	AVL
random1	100	13,12	7,6	7,6	13,11	4,4	4,4	8,7
	1,000	22,22	10,10	10,10	23,22	6,6	6,6	12,12
	10,000	32,31	14,14	14,14	33,32	8,8	8,8	16,16
	100,000	41,41	17,17	17,17	41,42	9,9	9,9	20,20
random2	100	13,13	7,7	7,7	14,12	4,4	4,4	8,7
	1,000	23,69	10,10	10,10	22,60	6,6	6,6	12,11
	10,000	38,93	14,14	14,14	35,82	8,7	8,7	16,15
	100,000	147,199	17,17	17,17	135,186	9,9	9,9	19,19
increasing	100	–	7,8	7,8	11,16	4,5	4,5	7,8
	1,000	–	10,11	10,11	24,27	6,7	6,7	10,11
	10,000	–	14,14	14,14	33,33	8,8	8,8	14,14
	100,000	–	17,17	17,17	46,43	9,9	9,9	17,17
decreasing	100	–	7,7	7,7	11,15	4,5	4,5	7,8
	1,000	–	10,10	10,10	24,21	6,7	6,7	10,11
	10,000	–	14,14	14,14	33,36	8,8	8,8	14,14
	100,000	–	17,17	17,17	46,43	9,9	9,9	17,17

n = the number of elements in initial data structures
Total number of operations performed = 100,000

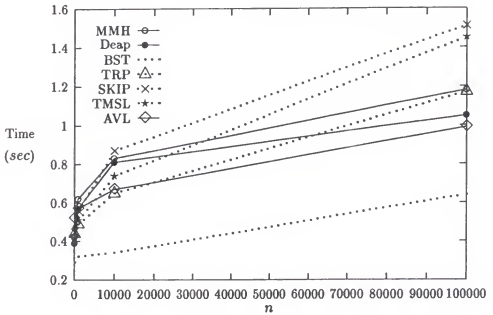


Figure 4.15. Run time on random1

Table 4.9. Run time using integer keys

inputs	n	BST	MMH	Deap	TRP	SKIP	TMSL	AVL
random1	100	0.29	0.42	0.39	0.44	0.45	0.42	0.52
	1,000	0.32	0.62	0.57	0.49	0.56	0.51	0.57
	10,000	0.34	0.83	0.81	0.65	0.87	0.74	0.67
	100,000	0.64	1.18	1.05	1.17	1.51	1.45	0.99
random2	100	0.27	0.42	0.39	0.47	0.51	0.46	0.56
	1,000	0.47	0.64	0.59	0.54	0.53	0.48	0.55
	10,000	0.59	0.85	0.78	0.72	0.89	0.71	0.69
	100,000	4.22	1.07	1.01	1.91	1.50	1.47	1.01
increasing	100	–	0.38	0.38	0.35	0.48	0.38	0.60
	1,000	–	0.60	0.63	0.42	0.65	0.53	0.53
	10,000	–	0.88	0.82	0.47	0.80	0.63	0.62
	100,000	–	1.12	1.15	0.57	0.92	0.85	0.77
decreasing	100	–	0.37	0.40	0.35	0.35	0.42	0.53
	1,000	–	0.63	0.62	0.43	0.33	0.38	0.55
	10,000	–	0.83	0.83	0.50	0.35	0.40	0.62
	100,000	–	1.05	1.10	0.63	0.40	0.45	0.83

Time Unit : *sec*
n = the number of elements in initial data structures
Total number of operations performed = 100,000

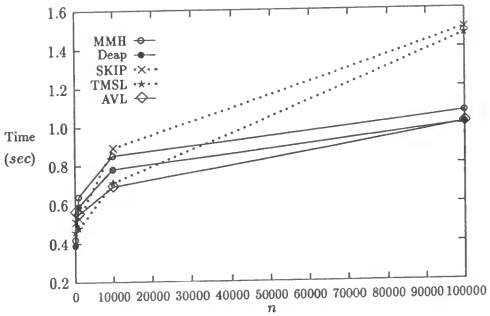


Figure 4.16. Run time on random2

a data structure for dictionaries. MSLs have the added advantage of using fixed size nodes.

Our experiments show that binary search trees (modified to handle equal keys) perform best of the tested double ended priority queue structures using random data. Of course, these are unsuitable for general application as they have very poor performance on ordered data. Min-max heaps, deaps and AVL trees guarantee $O(\log n)$ behavior per operation. Of these three, AVL trees generally do best for large n. It is possible that other balanced search structures such as bottom-up red-black trees might do even better. Treaps and skip lists are randomized structures with $O(\log n)$ expected complexity. Treaps were generally faster than skip lists (except for decreasing data) as double ended priority queues.

For single ended priority queues, if we exclude BSTs because of their very poor performance on ordered data, WBLTs did best on the data sets random1 and random2 (except when $n = 100,000$), and decreasing. Heaps did best on the remaining data sets. The probabilistic structures TRP, SKIP and TMSL were generally slower than WBLTs. When the ratio m/n (m = number of operations, n = average queue size) is large, WBLTs (and LTs) outperform heaps (and all other tested structures) as the binary trees constructed tend to be highly skewed to the left and the length of the rightmost path is close to one.

Our experimental results for single ended priority queues are in marked contrast to those reported in Gonnet and Baeza-Yates [11, p228] where leftist trees are reported to take approximately four times as much time as heaps. We suspect this difference in results is because of different programming techniques (recursion vs. iteration, dynamic vs. static memory allocation, etc.) used in Gonnet and Baeza-Yates [11] for the different structures. In our experiments, all structures were coded using similar programming techniques.

CHAPTER 5
CONCLUSIONS

We have considered the problem of joining a row of compacted cells and developed heuristics to stretch cells and river-route the nets so that the layout area is minimized. Our proposed heuristics were compared, experimentally, with Fang [8] and found to produce layouts with less area.

We have developed a new weight balanced search structure called β-BBST. Our experimental results show that BBSTs generally have the best search cost of the structures considered. We recommend β-BBSTs for environments where searches are done with much greater frequency than inserts and/or deletes. Based on our experiments, we conclude that AVL trees remain the best dictionary structure for general applications.

We have also proposed two simplified versions of the BBST called SBBST and BBSTD. Both versions are very competitive with BBSTs. The BBST and SBBST provide slightly better balance than provided by WB(α) trees.

Another new data structures, weight biased leftist trees, developed by us, has better performance than leftist trees as a data structure for single ended priority queues. We have also proposed a fixed node size representation for skip lists. As a data structure for dictionaries, this has a better performance than skip lists with variable node size.

APPENDIX A
ABBREVIATIONS

Table A.1. Abbreviations used in tables

AVL	Adelson-Velskii and Landis' trees
B-Heap	binomial heaps
BBST	β-balanced binary search trees
BBSTD	BBSTs without deletion
BST	unbalanced binary search trees
Deap	deaps
DSL	deterministic skip lists
Heap	heaps
LT	leftist trees
MMH	min-max heaps
MSL	modified skip lists
RB-B	bottom-up red-black trees
RB-T	top-down red-black trees
SBBST	simple β-balanced binary search trees
SKIP	skip lists
TMSL	threaded modified skip lists
TRP	treaps
WB	weight balanced trees
WBLT	weight biased leftist trees

REFERENCES

[1] ARAGON, C. R., AND SEIDEL, R. G. Randomized Search Trees. In *Proc. 30th Ann. IEEE Symposium on Foundations of Computer Science* (Oct. 1989), pp. 540–545.

[2] ATKINSON, M., SACK, J., SANTORO, N., AND STROTHOTTE, T. Min-max Heaps and Generalized Priority Queues. *Commun. ACM 29*, 10 (1986), 996–1000.

[3] BARATZ, A. Algorithms for Integrated Circuit Signal Routing. Ph.D. dissertation, 1981. MIT.

[4] BLUM, N., AND MEHLHORN, K. On the Average Number of Rebalancing Operations in Weight-balanced Trees. *Theoretical Comput. Sci. 11* (1980), 303–320.

[5] BOYER, D. G. Symbolic Layout Compaction Review. In *Proc. 25th Design Automation Conf.* (June 1988), pp. 383–389.

[6] CARLSON, B. S., AND LEE, S. J. Delay Optimization of Digital CMOS VLSI Circuits by Transistor Reordering. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 14*, 10 (Oct. 1995), 1183–1192.

[7] CARLSSON, S. The Deap: a Double-Ended Heap to Implement Double-Ended Priority Queues. *Inf. Process. Lett. 26* (1987), 33–36.

[8] CHENG, G., AND DESPAIN, A. Fang: A Joiner for Compacted Cells. In *VLSI 89* (1989), pp. 455–463.

[9] CRANE, C. Linear Lists and Priority Queues as Balanced Binary Trees. Tech. Rep. CS-72-259, Stanford University, 1972. Dept. of Comp. Sci.

[10] FREDMAN, M., AND TARJAN, R. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. *Journal of ACM 34*, 3 (1987), 596–615.

[11] GONNET, G. H., AND BAEZA-YATES, R. *Handbook of Algorithms and Data Structures*, 2nd ed. Addison-Wesley Pub. Co., Reading, MA, 1991.

[12] GUIBAS, L. J., AND SEDGEWICK, R. A Dichromatic Framework for Balanced Trees. In *Proc. 19th FOCS* (1978), pp. 8–21.

[13] HARRISON, A. J. VLSI Layout Compaction Using Radix Priority Search Trees. In *Proc. 28th Design Automation Conf.* (March 1991), pp. 732–735.

[14] HOROWITZ, E., AND SAHNI, S. *Fundamentals of Data Structures in Pascal*, 4th ed. W. H. Freeman and Company, New York, NY, 1994.

[15] LEISERSON, C., AND PINTER, R. Optimal Placement for River Routing. *SIAM J. Comput. 3*, 3 (Aug. 1983), 447–462.

[16] LIM, A. Efficient Algorithm for CAD in VLSI . Ph.D. dissertation, 1992. University of Minnesota, MN.

[17] LIM, A., CHENG, S., AND SAHNI, S. Optimal Joining of Compacted Cells. *IEEE Trans. Comput. 42*, 5 (May 1993), 597–607.

[18] MCCREIGHT, E. M. Priority Search Trees. *SIAM J. Comput. 14*, 2 (May 1985), 257–276.

[19] MIRZAIAN, A. River Routing in VLSI. *J. Comput. Syst. Sci. 34*, 1 (1987), 43–54.

[20] MUNRO, J. I., PAPADAKIS, T., AND SEDGEWICK, R. Deterministic Skip Lists. In *3rd Annual ACM-SIAM Symposium on Discrete Algorithms* (Jan. 1992), pp. 367–375.

[21] NIEVERGELT, J., AND REINGOLD, E. M. Binary Search Trees of Bounded Balance. *SIAM J. Comput. 2*, 2 (March 1973), 33–43.

[22] PAPADAKIS, T. Skip Lists and Probabilistic Analysis of Algorithms. Ph.D. dissertation, 1993. University of Waterloo.

[23] PINTER, R. Y. On Routing Two Point Nets Across A Channel. In *Proc. 19th Design Automation Conf.* (June 1982), pp. 899–902.

[24] PINTER, R. Y. River Routing: Methodology and Analysis. In *Third Caltech Conference on VLSI*, R. Bryant., Ed. Computer Science Press, Rockville, MD, March 1983, pp. 141–163.

[25] PREPARATA, F. P., AND W. LIPSKI, J. Optimal Three-Layer Channel Routing. *IEEE Trans. Comput. c-33*, 5 (May 1984), 427–437.

[26] PUGH, W. Skip Lists: a Probabilistic Alternative to Balanced Trees. *Commun. ACM 33*, 6 (1990), 668–676.

[27] SAHNI, S. *Software Development in Pascal.* NSPAN Printing and Publishing Co., Gainesville, FL, 1993.

[28] SEDGEWICK, R. *Algorithms in C++.* Addison-Wesley Pub. Co., Reading, MA, 1994.

[29] SHENOY, N., AND RUDELL, R. Efficient Implementation of Retiming. In *Proc. International Conf. on CAD* (June 1994), pp. 226–233.

[30] SHERWANI, N. *Algorithms for VLSI Physical Design Automation.* Kluwer Academic, Norwell, MA, 1992.

[31] TARJAN, R. E. Updating a Balanced Search Tree in O(1) Rotations. *Inf. Process. Lett. 16* (June 1983), 253–257.

[32] TOMPA, M. An Optimal Solution to A Wire-routing Problem. *J. Comput. Syst. Sci. 23*, 2 (May 1981), 127–150.

[33] WESTE, N. Virtual Grid Symbolic Layout. In *Proc. 18th Design Automation Conf.* (June 1981), pp. 225–233.

BIOGRAPHICAL SKETCH

Seonghun Cho was brought up in the Republic of Korea. He completed his undergraduate degree in mathematics from Seoul National University. He obtained an M.S degree in computer science from University of Missouri-Columbia, Columbia, MO. He started to work towards his Ph.D from Fall 1991 in the Computer and Information Science and Engineering Department at the University of Florida. His research interests include VLSI CAD algorithms and data structures. He will be graduating in May 1996.

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Sartaj Sahni, Chairman
Professor of Computer and
Information Science and Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Li-Min Fu
Associate Professor of Computer and
Information Science and Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Theodore Johnson
Associate Professor of Computer and
Information Science and Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Sanguthevar Rajasekaran
Associate Professor of Computer and
Information Science and Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Paul W. Chun
Professor of Biochemistry and
Molecular Biology

This dissertation was submitted to the Graduate Faculty of the College of Engineering and to the Graduate School and was accepted as partial fulfillment of the requirements for the degree of Doctor of Philosophy.

May 1996

Winfred M. Phillips
Dean, College of Engineering

Karen A. Holbrook
Dean, Graduate School